图灵程序设计丛书



Learning Data Mining with Python

Python数据挖掘入门与实践

【澳】Robert Layton◎著　杜春晓◎译

全面释放Python的数据分析能力，掌握大数据时代核心技术，
轻松入门数据挖掘技术并将其应用于实际项目

中国工信出版集团　人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Learning Data Mining with Python

Python数据挖掘入门与实践

【澳】Robert Layton◎著　杜春晓◎译

人民邮电出版社
北京

图书在版编目（C I P）数据

Python数据挖掘入门与实践 / （澳）罗伯特·莱顿（Robert Layton）著 ; 杜春晓 译. -- 北京 : 人民邮电出版社, 2016.7
（图灵程序设计丛书）
ISBN 978-7-115-42710-6

Ⅰ. ①P… Ⅱ. ①罗… ②杜… Ⅲ. ①数据处理软件—程序设计 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2016)第132392号

内 容 提 要

本书作为数据挖掘入门读物，介绍了数据挖掘的基础知识、基本工具和实践方法，通过循序渐进地讲解算法，带你轻松踏上数据挖掘之旅。本书采用理论与实践相结合的方式，呈现了如何使用决策树和随机森林算法预测美国职业篮球联赛比赛结果，如何使用亲和性分析方法推荐电影，如何使用朴素贝叶斯算法进行社会媒体挖掘，等等。本书也涉及神经网络、深度学习、大数据处理等内容。

本书面向愿意学习和尝试数据挖掘的程序员。

◆ 著　[澳] Robert Layton
译　杜春晓
责任编辑　朱　巍
执行编辑　谢婷婷　牛现云
责任印制　彭志环
◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　印刷
◆ 开本：800×1000　1/16
印张：15.75
字数：372千字　2016年 7 月第 1 版
印数：1 – 4 000册　2016年 7 月北京第 1 次印刷
著作权合同登记号　图字：01-2016-2261号

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

译　者　序

云计算、大数据、物联网，这几年很火。到现在为止，人们对云计算的激情已经回落到比较理智的水平，各种云基础设施已投入使用，支撑起关系国计民生的信息化应用。物联网还在建设中，家电智能化、个人健康信息数字化、交通智能化等趋势在我们身边悄然推进。开放互联的概念不再囿于传统的互联网思维，我们生活所触及的一切正在被编织到一张包罗万象的大网之中。它将会对社会产生何种影响，我们拭目以待。虽然大数据现在很火，各种大数据研究中心相继建立，但这只是刚刚开始。随着更多的人和设备接入互联网，随着人们对世界认识的加深和新工具的研发，数据规模将加速膨胀，超乎想象。大数据的春天才刚刚到来。数据采集能力上去之后，势必要求数据挖掘能力跟得上。正如作者在第12章中讲到的，大数据带来的一个挑战就是，重要信息可能被垃圾信息湮没。由此我们不难推断出数据挖掘技术在发现、突显和传承人类文明方面将起到不可替代的作用。本书讲解的正是大数据时代的核心技能——数据挖掘技术，可以预见该项技术将发挥出越来越重要的作用。

本书讲解了如何用Python语言进行数据挖掘。Python是一种通用型编程语言，它简单易学，上手快，有着丰富的第三方库，社区氛围友好。从数据采集、分析一直到应用开发层面，Python都有成熟的库。使用Python语言进行开发，无需过多关注语言细节，开发者可以将主要精力放到业务本身。书中使用IPython Notebook作为开发环境，它将代码执行、富文本、公式编辑、绘图、多媒体等功能集合在一起，是科学计算和数据分析的好工具。书中所涉及的数据挖掘对象很丰富，有Iris鸢尾花卉数据集、Ionosphere电离层数据集、NBA比赛结果、MovieLens电影评分数据集、古登堡计划所收集的图书、安然公司邮件数据集、博客语料、CIFAR-10图像数据集等。从这些分属于不同行业的数据集，也能一窥数据挖掘应用之广。此外，作者还介绍了从Twitter、Reddit网站采集数据的方法。在算法方面，除了常见的决策树、朴素贝叶斯、支持向量机等，作者还介绍了最近几年非常热的深度学习。大数据、深度学习对计算能力要求很高，作者介绍了如何在亚马逊云主机上运行MapReduce任务。这本书由浅入深，以真实数据为研究对象，逐渐增大数据集规模，真刀实枪地向读者介绍了Python数据挖掘是怎么回事，并给读者进一步学习指出了多种可能的方向。工程实践之余，作者还不忘介绍数据挖掘常用思路，毫不保留地把自己积攒的宝贵经验传授给读者。这一点我在阅读过程中，深有体会。正如作者自己在前言里所写的，书中不会涉及大量公式推导，所有的算法都是以很直观的形式向读者介绍，所以即使你缺乏一定的数学基础，只要肯用功，也不用担心自己读不懂。

回到七八年前，当我还是一名英语专业学生的时候，我压根不会想到有一天会学编程，会去翻译这样一本书。后来有幸读了计算机辅助翻译这样一个专业，才开始接触到计算机知识，但是当父亲跟我提起数据建模时，我还是一脸茫然。研究生几年，系里为我们这些非计算机背景的学生开设了Python编程课。从那时起我就有事没事学点Python，一开始是照着*Natural Language Processing with Python*的示例敲，自那时起五年之后我竟想起给NLTK提交几处微小的改动。大约是为了激励我这个后生继续为他们服务，"居心叵测"的Steven Bird竟把我加入到贡献者名单里。去中关村图书大厦的时候，我常常喜欢浏览一下语言与程序设计书架上有没有关于Python的新书，碰到喜欢的就翻翻看，这几年眼看着Python书多了起来，很是欣慰。此外，我去北大、北外旁听过计算语言学、概率统计等课程，去北航旁听过计算机系统基础，看过Udacity的统计学入门和吴恩达老师的机器学习课程视频，兴致来了也曾捧着Rosen的《离散数学及其应用》读上几页。工作中，经常帮同事写个简单的Python程序处理数据，最近还帮他们爬取了一个网站。PyCon北京，我连着去了三四届了，每次都有或多或少的收获，2015年我见过一位大神行云流水般演示用pandas处理数据，很受震撼。以上就是我与Python、数据挖掘的交集。我想说的是，不要再用上学时读的那个专业的思维局限自己的发展，学科的界限在模糊，融合的趋势在增强，数学的重要性在提升。提到数学，今天还看了一个TED演讲视频，说的是借助计算机改变传统数学教育方法。这种理念什么时候能应用到一线教学，非常值得期待。生在这个充满变革的时代，倍感幸运。

以我有限的水平去翻译这样一本书，心里不免焦虑。遇到问题，我四处寻找能人相助。感谢作者Robert Layton，我每次发邮件向他求教或确认问题，他总能很快地回复我，有时第二天回复，还会说抱歉回复晚了。感谢我的同学黄子轩、孙伟、周星，他们在我学习计算机科学的路上给予了很多指导和帮助。翻译本书时，我还向子轩求证作者在第6章给出的示例是否有误。感谢上海大学研究生钱亦欣同学，他帮我审读了第3章，并给出若干很专业的修改意见，上述第6章那处问题，我也曾向他请教过，最终证明是原书弄错了。感谢李少华，他帮我弄明白了Python在Windows系统中的环境变量设置方法。感谢陈健锁，我曾就数据库相关术语向他请教。感谢图灵的朱巍编辑，是她促成了我最终去翻译这样一本趣味盎然的书！感谢图灵公司，我是你们忠实的读者！最后感谢我的妻子，她承担了照顾女儿的重任。女儿的出生，让我惊叹于生命的奇妙！感谢岳父岳母一家人帮忙照看孩子，我才有时间去做翻译。感谢我的父亲、姐姐，他们以我翻译本书为骄傲。

由于本人学识有限，且时间仓促，书中翻译错误、不当和疏漏之处在所难免，望读者批评指正。

杜春晓

2016年1月3日

前　言

你是不是向往数据挖掘的殿堂，却不得其门而入？如果是的话，这本书就是为你而写的。

很多讲授数据挖掘的书涵盖了大量数学知识，倘若读者有较好的数学背景，这自然不错，但我觉得这些书往往只见树木不见森林；也就是说，它们过于关注算法的工作原理，而忘记了我们使用这些算法的初衷。

本书的目标读者是具备一定编程能力、渴望学习数据挖掘的人。我的目标是，如果你认真学完本书，能较好地理解数据挖掘的基础知识，掌握用数据挖掘知识解决问题的最佳实践，此外还能从书中找到几个值得你深入研究的方向。

本书的每一章都会介绍一个新的主题，我会给出该主题的相关算法和数据集。因此，各章主题之间跳跃有点大，阅读本书的过程中需要你的大脑能快速切换。每学完一章，你都要思考一下有没有什么办法能够提升该章中算法的效果，然后尝试去实现它！

我最喜欢的格言中有一条出自莎士比亚的《亨利四世》[①]：

但是你叫他们的时候，他们真的会来吗？

在剧中，有人声称自己会招魂，于是霍茨波（Hotspur）回复了上面这句话。霍茨波的意思是每个人都会招魂，但问题是招过之后，魂来不来。

跟招魂类似，学习数据挖掘无外乎做实验，获得结果。实现一个新的数据挖掘算法或是改善现有实验结果的想法，每个人都能提出来，但真正重要的是你能不能实现它，还有结果是否尽如人意。

本书主要内容

第1章　开始数据挖掘之旅，介绍我们即将用到的技术，接着通过讲解两个基础算法的实现方法达到热身目的。

---

①《亨利四世》是莎士比亚所著的历史剧，分为上下两篇。接下来这句台词出自上篇第三幕第一场，原文为“But will they come when you do call for them?”。遵照朱生豪先生的译法，将“Hotspur”译为“霍茨波”。——译者注

第2章 用`scikit-learn`估计器分类，涵盖了数据挖掘的一个重要主题——分类。这一章还会介绍将数据挖掘流程标准化的流水线结构，便于你管理实验流程。

第3章 用决策树预测获胜球队，介绍决策树和随机森林两个新算法。我们将通过抽取区分度高的特征来预测获胜选手。

第4章 用亲和性分析方法推荐电影，思考根据以往消费记录推荐产品的问题，介绍Apriori算法。

第5章 用转换器抽取特征，介绍不同类别特征的抽取方法及不同数据集的处理方法。

第6章 使用朴素贝叶斯进行社会媒体挖掘，使用朴素贝叶斯算法自动分析来自社交网站Twitter的文本信息。

第7章 用图挖掘找到感兴趣的人，采用聚类和网络分析方法，发现社会媒体上感兴趣的人。

第8章 用神经网络破解验证码，从图像中抽取信息，然后训练神经网络，用来发现图像中的单词和字母。

第9章 作者归属问题，通过抽取文本特征，使用支持向量机算法，找出文档的作者。

第10章 新闻语料分类，使用k-means聚类算法，根据新闻文章内容进行分类。

第11章 用深度学习方法为图像中的物体进行分类，采用深度神经网络算法确定图像中的物体。

第12章 大数据处理，探讨对大数据进行数据挖掘的流程及方法。

附录 依次介绍各章的参考资料，便于读者深入了解各章内容。

本书的阅读前提

毫无疑问，要完成本书的学习，你需要准备一台电脑。电脑不能太旧，但也不用配置太高。只要是最近几年的处理器（2010年以后），4GB内存（RAM）就足够了。在性能稍差一点的机器上运行本书中的大部分代码，应该也不会有问题。

最后两章的代码对机器性能要求较高。在这两章中，我介绍了如何使用亚马逊网络服务（AWS）运行代码。这多少可能会让你破费一些，与在本地运行代码相比，好处是不用做那么多系统配置。如果你不想掏钱购买亚马逊的网络服务，可以在本地主机上配置好所有的工具，当然你需要一台配置较高的计算机：2012年及以后的处理器，内存在4GB以上。

我推荐使用Ubuntu操作系统，但是本书示例代码也可以在Windows、Mac和其他任何Linux系统上运行。然而，你可能需要参考系统的相关文档来完成必要工具的安装。

本书中，我将使用命令行工具`pip`来安装我们用到的Python库。你也可以使用Anaconda，下载地址为http://continuum.io/downloads。

本书所有代码都已在Python 3中测试过。大部分代码无需修改就能在Python 2下运行。如果你遇到什么问题解决不了，请发邮件给我们，我们帮你看看怎么解决。

本书的目标读者

本书是写给那些想把数据挖掘技术应用到实际项目中，却不知道怎么开始的程序员。

如果你没有编程经历，我强烈建议你在学习本书之前，至少先了解一下编程的基础知识。本书直接略过这部分内容，也不会把过多精力放在代码的具体实现上。也就是说，简要学习一下编程基础知识再来学习本书就行。本书不对你的编程技能做过高要求，所以你没必要先成为编程高手！

我强烈建议在阅读本书前最好先积累一些Python编程经验。如果没有的话，也没关系，但是你可能需要先瞧瞧Python代码是怎么回事，可能的话先看看IPython Notebook的教程。用IPython Notebook写程序，跟用其他编辑器写程序（比如使用功能全面的IDE编写Java程序）有一定区别。

排版约定

本书使用不同的文本样式来区分不同类别的内容。以下是常用样式及其用途说明。

首先，我们来看一下最重要的代码所使用的样式。

```
if True:
    print("Welcome to the book")
```

请留意缩进。缩进在Python中表示层级的概念。本书中，我们使用四个空格来表示缩进。你可以按照自己的喜好，使用不同数量的空格（或制表符），但是应该保持一致。如果你被缩进搞糊涂了，请参考本书的示例代码。

在正文中引用代码时，会使用下面这种样式：`I'll use this format`。你不需要在IPython Notebook中输入这些代码，除非我在正文中明确要求你这么做。

所有的命令行输入或输出都使用下面这种样式：

```
# cp file1.txt file2.txt
```

新的术语和重要的词语使用*楷体*。

此图标表示警告或重要信息。

此图标表示提示或技巧。

读者反馈

我们热忱地欢迎读者朋友给予我们反馈，告诉我们你对于这本书的所思所想——你喜欢或是不喜欢哪些内容。大家的反馈对我们来说至关重要，将能帮助我们确定到底哪些内容是读者真正需要的。

如果你有一般性建议的话，请发邮件至feedback@packtpub.com，请在邮件主题中写清书的名称。

如果你是某一方面的专家，对某个主题特别感兴趣，有意向自己或是与别人合作写一本书，请到www.packtpub.com/authors查阅我们为作者准备的帮助文档。

客户支持

为自己拥有一本Packt出版的书而自豪吧！为了让你的书物有所值，我们还为你准备了以下内容。

下载示例代码

如果你是从http://www.packtpub.com网站购买的图书，用自己的账号登录后，可以下载所有已购图书的示例代码。如果你是从其他地方购买的，请访问http://www.packtpub.com/support网站并注册，我们会用邮件把代码文件直接发给你。

下载配套 PDF 文件

我们还为你准备了一个PDF文件，该文件包含书中的所有屏幕截图/图表。这些彩色图像能弥补图书中黑白图像的不足，有助于你理解本书内容。该文件的下载地址为https://www.packtpub.com/sites/default/files/downloads/6053OS_ColorImages.pdf。

勘误表

即使我们竭尽所能来保证图书内容的正确性，错误也在所难免。如果你在我们出版的任何一本书中发现错误——可能是在文本或代码中——倘若你能告诉我们，我们将会非常感激。你的善举足以减少其他读者在阅读出错位置时的纠结和不快，帮助我们在后续版本中更正错误。如果你发现任何错误，请访问http://www.packtpub.com/submit-errata，选择相应书籍，点击“Errata Submission Form”链接，输入错误之处的具体信息。你提交的错误得到验证之后，我们就会接受你的建议，该处错误信息将会上传到我们的网站或是添加到已有勘误表的相应位置。

访问https://www.packtpub.com/books/content/support，在搜索框中输入书名，可查看该图书已有的勘误信息。这部分信息会在Errata部分显示。

侵权

所有媒体在互联网上都面临的一个问题就是侵权。对Packt来说，我们严格保护我们的版权和许可证。如果你在网上发现针对我们出版物的任何形式的盗版产品，请立即告知我们地址或网站名称，以便我们进行补救。

请将盗版书籍的网站地址发送到copyright@packtpub.com。

如果你能这么做，就是在保护我们的作者，保护我们，只有这样我们才能继续以优质内容回馈像你这样热心的读者。

问题

你对本书有任何方面的问题，都可以通过questions@packtpub.com邮箱联系我们，我们也将尽最大努力来帮你答疑解惑。

目　录

第 1 章

开始数据挖掘之旅

1

我们今天的数据采集规模在人类历史上是空前的，日常生活也越来越依赖我们所采集的这些信息。我们希望计算机能把网页翻译成其他语言，预报天气，推荐我们喜欢的书，诊断我们的健康问题。类似的需求还会继续增长，我们会需要更多的应用和更高的精确性。数据挖掘技术可以用来训练计算机，使其根据已有数据做出决策。如今，数据挖掘技术已成为支撑很多高科技系统的骨干。

Python的迅速普及并非偶然。它的灵活度高；模块众多，可以执行很多任务；比起其他任何编程语言，Python代码通常更为简洁，可读性更强。Python在数据挖掘领域已经形成了一个由研究员、从业者和新手组成的氛围活跃的大社区。

本章将介绍如何使用Python进行数据挖掘，主要会涉及以下几个主题。

- 数据挖掘简介及其应用场景
- 搭建Python数据挖掘环境
- 亲和性分析示例：根据购买习惯推荐商品
- （经典）分类问题示例：根据测量结果推测植物的种类

1.1 数据挖掘简介

数据挖掘旨在让计算机根据已有数据做出决策。决策可以是预测明天的天气、拦截垃圾邮件、检测网站的语言，或者在约会网站上发现新的恋爱对象。数据挖掘方面的应用已经有很多，新的应用也在源源不断地出现。

数据挖掘涉及算法、统计学、工程学、最优化理论和计算机科学相关领域的知识。除此之外，我们还会用到语言学、神经科学、城市规划等其他领域的概念或知识。要想充分发挥数据挖掘的威力，通常需要在算法中整合这些属于特定领域的知识。

虽然数据挖掘相关应用的实现细节可能千差万别，但是从较高的层次看，它们往往大同小异。数据挖掘的第一步一般是创建数据集，数据集能够描述真实世界的某一方面。数据集主要包括以下两个部分。

- 表示真实世界中物体的样本。样本可以是一本书，一张照片，一个动物，一个人或是其他任何物体。
- 描述数据集中样本的特征。特征可以是长度、单词频率、腿的数量、创建时间等。

接下来是调整算法。每种数据挖掘算法都有参数，它们或者是算法自身包含的，或者是使用者添加的。这些参数会影响算法的具体决策。

举个简单的例子，我们希望计算机能够把人按照个子高矮分成两大类。我们首先采集数据，得到包含每个人身高的一组数据，以及对他们高矮的判断。

人	身　高	高还是矮
1	155cm	矮
2	165cm	矮
3	175cm	高
4	185cm	高

接下来要做的就是调整我们的算法。作为一个简单的算法，如果身高高于*x*，我们就认为这个人是高个子，否则，他就属于矮个子。我们的算法要过一遍数据，确定*x*的最佳值。对于上面的数据集，*x*比较合理的值为170cm。任何高于170cm的人就被归到高个子一类中，其余则为矮个子。

在上面这个数据集中，特征显而易见为身高。因为我们想知道人们的高矮，所以采集了他们的身高数据。抽取特征是数据挖掘过程的一个重要环节。本书后面的章节中会介绍从数据集中抽取区分度高的特征的方法。特征抽取往往需要对相关领域有着深入的理解，或至少需要多次试错。

> 本书中使用Python语言介绍数据挖掘。出于讲解的需要，为了保证代码、流程的清晰易懂，我们有时候跳过了能够提升算法速度、效果的细节，没有采用最优方案。

1.2　使用 Python 和 IPython Notebook

本节将介绍Python的安装方法，及本书所用到的开发环境IPython Notebook的搭建方法。此外，还将安装第一部分示例代码所用到的`numpy`库。

1.2.1　安装 Python

Python是一门出色的、应用范围广泛且简单易用的编程语言。

本书将使用Python 3.4版本，你可以根据自己的系统从Python官网https://www.python.org/downloads/下载合适的版本。

Python主要有两大版本Python 3.4和Python 2.7。记得要下载安装Python 3.4，本书所有代码都

在该版本中测试过。

本书假定读者了解编程和Python相关知识。本书不要求你是Python编程高手，当然有较多的知识储备学起来更容易。

如果你没有任何编程经验，我建议你先看看《Python学习手册》。

Python官网为新手准备了两份在线教程。

- 非程序员背景，想通过Python学习编程：
  https://wiki.python.org/moin/BeginnersGuide/NonProgrammers
- 程序员背景，想专门学习Python：
  https://wiki.python.org/moin/BeginnersGuide/Programmers

> Windows用户设置好环境变量后，才能在命令行中使用Python。方法如下。首先，找到Python 3的安装路径，默认为C:\Python34。接下来，在命令行（cmd程序）中输入以下命令：将环境设置为`PYTHONPATH=%PYTHONPATH%; C:\Python34`[1]。如果你将Python安装到其他路径下，请根据实际情况调整上述命令中的C:\Python34。

安装好Python，打开命令提示符，输入以下命令：

```
$ python3
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on Linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, world!")
Hello, world!
>>> exit()
```

请注意，我们用美元符号（$）表示紧跟在后面的语句是一条命令，要输入到终端（Unix系统中的shell，Windows系统中的cmd程序）。美元符号及后面的空格无须输入。输入后面的内容，然后敲回车执行命令。

运行完经典的“Hello, world!”例子后，退出Python，继续安装用来运行Python代码的更为高级的开发环境IPython Notebook。

> Python 3.4内置了Python的包管理器pip，用它来安装Python包很方便。使用`$ pip3 freeze`[2]命令可以验证pip是否能正常运行，该命令还会输出你用它安装过哪些包。

① Python官网介绍了Windows系统的两种环境变量设置方法。建议直接把Python的安装路径添加到Path中，位置如下：计算机-属性-高级系统设置-环境变量，这也是官网介绍的第一种方法。译者使用的就是这一种。作者讲的是第二种方法，详见https://docs.python.org/3.4/using/windows.html#excursus-setting-environment-variables。——译者注

② Windows用户需要事先把pip添加到环境变量中，才能在命令行使用。——译者注

1.2.2　安装 IPython

Python开发平台IPython提供多种Python开发工具和开发环境，比标准解释器多出好多功能。IPython Notebook功能强大，有了它，你就可以在Web浏览器中编写程序。它会为代码添加样式，显示运行结果，允许你为代码添加注释。用它来做数据分析再好不过，我们将把它作为主要的开发环境。

请在命令提示符后（注意不是Python中），输入以下命令安装IPython：

```
$ pip install ipython[all]
```

如果要为系统所有用户安装IPython，需要管理员权限。如果你只想自己用或者没有权限做系统级别的变更，则使用以下命令为当前用户安装即可：

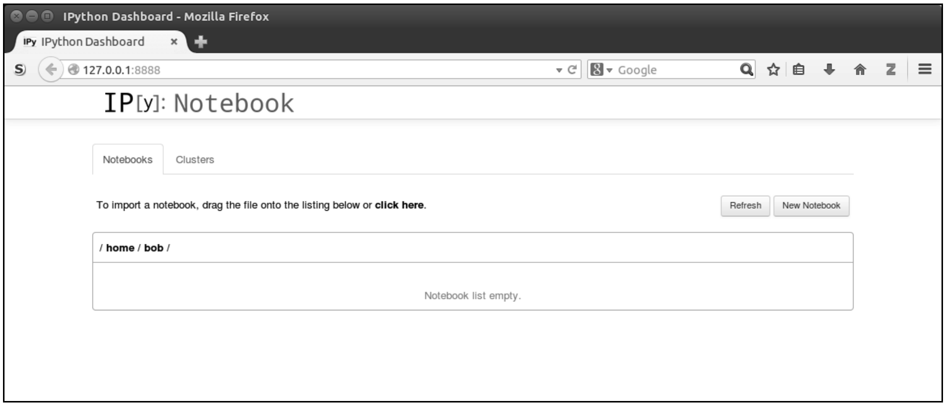
```
$ pip install --user ipython[all]
```

以上命令只为当前用户安装IPython——该系统的其他用户将无法使用。安装过程中若遇到问题，请查阅官方文档，了解更多帮助信息：http://ipython.org/install.html。

安装好IPython Notebook后，运行方式如下：

```
$ ipython3 notebook
```

上述命令帮你做了两件事。首先，在命令提示符界面创建一个IPython Notebook实例。其次，打开Web浏览器，连接到实例，你可以在此创建新的笔记本文件[①]。Notebook界面如下图所示（注意图中的home/bob为当前用户的主目录，你看到的是自己的主目录，所以目录名称很可能不同）。



IPython Notebook的关闭方法如下：打开运行实例的终端界面（就是你之前用IPython命令创建Notebook实例的界面），按下Ctrl+C键，系统提示`Shutdown this notebook server (y/[n])?`，询问你是否关闭笔记本服务器。输入`y`，敲回车，`IPython Notebook`就会关闭。

① 笔记本文件，英文为“notebook”，即用IPython Notebook创建的文件。——译者注

1.2.3 安装 scikit-learn 库

scikit-learn是用Python开发的机器学习库，它包含大量机器学习算法、数据集、工具和框架。它以Python科学计算的相关工具集为基础，其中numpy和scipy等都针对数据处理任务进行过优化，因此scikit-learn速度快、扩展性强，新手会觉得它很好用，老手也不会觉得它功能逊色。更多内容请见第2章。

scikit-learn库可用Python 3提供的pip工具进行安装，之前没有安装NurmPy和SciPy的话，也会顺便安装。用管理员/根用户权限打开一个终端，然后输入以下命令：

```
$ pip3 install -U scikit-learn
```

> Windows用户在安装scikit-learn之前，可能需要先安装NurmPy和SciPy。安装指南请见www.scipy.org/install.html。

Ubuntu或红帽（Red Hat）等Linux系统的用户也许希望用自带的包管理器安装scikit-learn，但是它们提供的版本很可能不是最新的，所以在安装前需仔细核对版本。本书使用的版本不能低于0.14，否则书中代码可能无法运行。

如何通过编译源文件进行安装，以及更多的安装指南，请见官方文档：

http://scikit-learn.org/ stable/install.html。

1.3 亲和性分析示例

终于迎来了第一个数据挖掘的例子，我们拿这个亲和性分析的示例来具体看下数据挖掘到底是怎么回事。数据挖掘有个常见的应用场景，即顾客在购买一件商品时，商家可以趁机了解他们还想买什么，以便把多数顾客愿意同时购买的商品放到一起销售以提升销售额。当商家收集到足够多的数据时，就可以对其进行亲和性分析，以确定哪些商品适合放在一起出售。

1.3.1 什么是亲和性分析

亲和性分析根据样本个体（物体）之间的相似度，确定它们关系的亲疏。亲和性分析的应用场景如下。

- 向网站用户提供多样化的服务或投放定向广告。
- 为了向用户推荐电影或商品，而卖给他们一些与之相关的小玩意。
- 根据基因寻找有亲缘关系的人。

亲和性有多种测量方法。例如，统计两件商品一起出售的频率，或者统计顾客购买了商品1后再买商品2的比率。当然还有别的方法，比如后面章节要讲的计算个体之间的相似度。

1.3.2 商品推荐

商品销售从线下搬到线上后，很多之前靠人工完成的工作只有实现自动化，才有望将生意做大。以向上销售为例，向上销售出自英文up-selling，指的是向已经购买商品的顾客推销另一种商品。原来线下由人工来完成的商品推荐工作，现在依靠数据挖掘技术就能完成，而且每年能为商家多进账几亿美元，强力助推电子商务革命的发展！

我们一起看下简单的商品推荐服务，它背后的思路其实很好理解：人们之前经常同时购买的两件商品，以后也很可能会同时购买。该想法确实很简单吧，可这就是很多商品推荐服务的基础，无论线上还是线下。

这种想法很容易转化为算法。顾客购买商品后，在向他们推荐商品前，先查询一下历史交易数据，找到以往他们购买同样商品的交易数据，看看同时购买了什么，再把它们推荐给顾客即可。该算法实际表现也不错，至少比随机推荐商品更有效。然而，它还有很大的提升空间，这正是数据挖掘一展身手的好机会。

为了简化代码，方便讲解，我们只考虑一次购买两种商品的情况。例如，人们去超市既买了面包，又买了牛奶。作为数据挖据入门性质的例子，我们希望得到下面这样的规则：

如果一个人买了商品X，那么他很有可能购买商品Y。

多件商品的规则会更为复杂，比如购买香肠和汉堡包的顾客比起其他顾客更有可能购买番茄酱，本书中不涉及这样的规则。

1.3.3 在 NumPy 中加载数据集

下载本书配套代码包，保存到你的计算机上，然后找到这个例子的数据集。本例中，建议你新建一个文件夹，把数据集和代码都放进去。在当前目录[①]下，启动IPython Notebook，导航进入新建的文件夹，创建一个新的笔记本文件。

处理该数据集要用到NumPy的二维数组，书中大部分例子都会用到这种数据结构。数组看上去像是一张表，每一行表示样本中一个个体，每一列表示一种特征。

数组的每一项为个体的某项特征值。说起来有些拗口，为方便讲解，使用如下代码把数据集加载进来，稍后输出数组的部分数据看看效果：

```
import numpy as np
dataset_filename = "affinity_dataset.txt"
X = np.loadtxt(dataset_filename)
```

① 在命令行，切换到新建的文件夹，输入ipython3 notebook命令。——译者注

运行IPython Notebook，创建笔记本文件，在第一个格子中输入上述代码。按下Shift+Enter（同时创建新的格子）运行代码。代码运行完毕后，第一个格子左侧的方括号中出现一个表示序号的数字，看到这个数字就表明程序运行结束。第一个格子应该如下所示：

```
In [1]: import numpy as np
        dataset_filename = "affinity_dataset.txt"
        X = np.loadtxt(dataset_filename)
```

对于笔记本文件，前面的代码运行完后，后面的才能运行；还没有轮到它运行或是在运行中时，方括号中显示一个星号。运行结束后，星号立刻变为序号。

记得把数据集文件和笔记本文件放到同一目录下。否则，请修改上述代码中`dataset_filename`变量的值。

接下来，我们看看数据集到底是什么样子。在笔记本空格子中输入以下代码，输出数据集的前5行看看：

```
print(X[:5])
```

> 如果你从http://www.packtpub.com网站购买的图书，登录后即可下载已购图书的代码文件。如果你是从别处购买的图书，访问http://www.packtpub.com/support，注册后，我们可以用电子邮件把你需要的文件发给你。[①]

上述代码的运行结果为前5次交易中顾客都买了什么。

```
In [2]: print(X[:5])

        [[ 0.  0.  1.  1.  1.]
         [ 1.  1.  0.  1.  0.]
         [ 1.  0.  1.  1.  0.]
         [ 0.  0.  1.  1.  1.]
         [ 0.  1.  0.  0.  1.]]
```

输出结果从横向和纵向看都可以。横着看，每次只看一行。第一行`(0, 0, 1, 1, 1)`表示第一条交易数据所包含的商品。竖着看，每一列代表一种商品。在我们这个例子中，这五种商品分别是面包、牛奶、奶酪、苹果和香蕉。从第一条交易数据中，我们可以看到顾客购买了奶酪、苹果和香蕉，但是没有买面包和牛奶。

每个特征只有两个可能的值，1或0，表示是否购买了某种商品，而不是购买商品的数量。1表示顾客至少买了1个单位的该商品，0表示顾客没有买该种商品。

① 注册后，可自行下载。——译者注

1.3.4 实现简单的排序规则

正如之前所说，我们要找出“如果顾客购买了商品X，那么他们可能愿意购买商品Y”这样的规则[①]。简单粗暴的做法是，找出数据集中所有同时购买的两件商品。找出规则后，还需要判断其优劣，我们挑好的规则用。

规则的优劣有多种衡量方法，常用的是支持度（support）和置信度（confidence）。

支持度指数据集中规则应验的次数，统计起来很简单。有时候，还需要对支持度进行规范化，即再除以规则有效前提下的总数量。我们这里只是简单统计规则应验的次数。

支持度衡量的是给定规则应验的比例，而置信度衡量的则是规则准确率如何，即符合给定条件（即规则的“如果”语句所表示的前提条件）的所有规则里，跟当前规则结论一致的比例有多大。计算方法为首先统计当前规则的出现次数，再用它来除以条件（“如果”语句）相同的规则数量。

接下来，通过一个例子来说明支持度和置信度的计算方法，我们看一下怎么求“如果顾客购买了苹果，他们也会购买香蕉”这条规则的支持度和置信度。

如下面的代码所示，通过判断交易数据中`sample[3]`的值，就能知道一个顾客是否买了苹果。这里，sample表示一条交易信息，也就是数据集里的一行数据。

```
In [9]: # First, how many rows contain our premise: that a person is buying apples
        num_apple_purchases = 0
        for sample in X:
            if sample[3] == 1:  # This person bought Apples
                num_apple_purchases += 1
        print("{0} people bought Apples".format(num_apple_purchases))

        36 people bought Apples
```

同理，检测`sample[4]`的值是否为1，就能确定顾客有没有买香蕉。现在可以计算题目给定规则在数据集中的出现次数，从而计算置信度和支持度。

我们需要统计数据集中所有规则的相关数据。首先分别为规则应验和规则无效这两种情况创建字典。字典的键是由条件和结论组成的元组，元组元素为特征在特征列表中的索引值，不要用实际特征名，比如“如果顾客购买了苹果，他们也会买香蕉”就用(3, 4)表示。如果某个个体的条件和结论均与给定规则相符，就表示给定规则对该个体适用，否则如果通过给定条件推出的结论与给定规则的结论不符，则表示给定规则对该个体无效。

为了计算所有规则的置信度和支持度，首先创建几个字典，用来存放计算结果。这里使用`defaultdict`，好处是如果查找的键不存在，返回一个默认值。需要统计的量有规则应验、规

① 一条规则由前提条件和结论两部分组成。——译者注

则无效及条件相同的规则数量。

```
from collections import defaultdict
valid_rules = defaultdict(int)
invalid_rules = defaultdict(int)
num_occurances = defaultdict(int)
```

计算过程需要用到循环结构，依次对样本的每个个体及个体的每个特征值进行处理。第一个特征为规则的前提条件——顾客购买了某一种商品。

```
for sample in X:
  for premise in range(4):
```

检测个体是否满足条件，如果不满足，继续检测下一个条件。

```
    if sample[premise] == 0: continue
```

如果条件满足（即值为1），该条件的出现次数加1。在遍历过程中跳过条件和结论相同的情况，比如“如果顾客买了苹果，他们也买苹果”，这样的规则没有多大用处。

```
    num_occurances[premise] += 1
    for conclusion in range(n_features):
        if premise == conclusion: continue
```

如果规则适用于个体，规则应验这种情况（valid_rules字典中，键为由条件和结论组成的元组）增加一次，反之，违反规则情况（invalid_rules字典中）就增加一次。

```
    if sample[conclusion] == 1:
      valid_rules[(premise, conclusion)] += 1
      else:
      invalid_rules[(premise, conclusion)] += 1
```

得到所有必要的统计量后，我们再来计算每条规则的支持度和置信度。如前所述，支持度就是规则应验的次数。

```
support = valid_rules
```

置信度的计算方法类似，遍历每条规则进行计算。

```
confidence = defaultdict(float)
for premise, conclusion in valid_rules.keys():
    rule = (premise, conclusion)
    confidence[rule] = valid_rules[rule] / num_occurances[premise]
```

我们得到了支持度字典和置信度字典，分别包含每条规则的支持度和置信度。我们再来声明一个函数，接收的参数有：分别作为前提条件和结论的特征索引值、支持度字典、置信度字典以及特征列表。输出每条规则及其支持度和置信度，对输出进行格式化，以方便查看。

```
def print_rule(premise, conclusion,
              support, confidence, features):
```

之前建立的features列表派上用场了，每条规则的条件、结论就是用features列表中特征的索引来表示的。输出时，把索引替换成相应的特征，更容易读懂。

```
premise_name = features[premise]
conclusion_name = features[conclusion]
print("Rule: If a person buys {0} they will also buy
  {1}".format(premise_name, conclusion_name))
```

接着输出规则的支持度和置信度。

```
print(" - Support: {0}".format(support[(premise,
                                        conclusion)]))
print(" - Confidence: {0:.3f}".format(confidence[(premise,
                                                conclusion)]))
```

写完后，自己测试一下代码是否可用——尝试更换条件和结论，看看输出结果如何。

```
In [31]: premise = 1
         conclusion = 3
         print_rule(premise, conclusion, support, confidence, features)

         Rule: If a person buys milk they will also buy apples
          - Confidence: 0.196
          - Support: 9
```

1.3.5　排序找出最佳规则

得到所有规则的支持度和置信度后，为了找出最佳规则，还需要根据支持度和置信度对规则进行排序，我们分别看一下这两个标准。

要找出支持度最高的规则，首先对支持度字典进行排序。字典中的元素（一个键值对）默认为没有前后顺序；字典的items()函数返回包含字典所有元素的列表。我们使用itemgetter()类作为键，这样就可以对嵌套列表进行排序。itemgetter(1)表示以字典各元素的值（这里为支持度）作为排序依据，reverse=True表示降序排列。

```
from operator import itemgetter
sorted_support = sorted(support.items(), key=itemgetter(1), re
verse=True)
```

排序完成后，就可以输出支持度最高的前5条规则。

```
for index in range(5):
    print("Rule #{0}".format(index + 1))
    premise, conclusion = sorted_support[index][0]
    print_rule(premise, conclusion, support, confidence, features)
```

结果如下所示：

```
In [40]: for index in range(5):
             print("Rule #{0}".format(index + 1))
             (premise, conclusion) = sorted_support[index][0]
             print_rule(premise, conclusion, support, confidence, features)

         Rule #1
         Rule: If a person buys cheese they will also buy bananas
          - Confidence: 0.659
          - Support: 27

         Rule #2
         Rule: If a person buys bananas they will also buy cheese
          - Confidence: 0.458
          - Support: 27

         Rule #3
         Rule: If a person buys apples they will also buy cheese
          - Confidence: 0.694
          - Support: 25

         Rule #4
         Rule: If a person buys cheese they will also buy apples
          - Confidence: 0.610
          - Support: 25

         Rule #5
         Rule: If a person buys bananas they will also buy apples
          - Confidence: 0.356
          - Support: 21
```

同理，我们还可以输出置信度最高的规则。首先根据置信度进行排序。

```
sorted_confidence = sorted(confidence.items(), key=itemgetter(1),
reverse=True)
```

再次输出看看结果。注意输出方法相同，但是请留意下面第三行代码里sorted_confidence的变化，不要继续使用sorted_support。

```
for index in range(5):
    print("Rule #{0}".format(index + 1))
    premise, conclusion = sorted_confidence[index][0]
    print_rule(premise, conclusion, support, confidence, features)
```

```
In [42]: for index in range(5):
             print("Rule #{0}".format(index + 1))
             (premise, conclusion) = sorted_confidence[index][0]
             print_rule(premise, conclusion, support, confidence, features)

Rule #1
Rule: If a person buys apples they will also buy cheese
 - Confidence: 0.694
 - Support: 25

Rule #2
Rule: If a person buys cheese they will also buy bananas
 - Confidence: 0.659
 - Support: 27

Rule #3
Rule: If a person buys bread they will also buy bananas
 - Confidence: 0.630
 - Support: 17

Rule #4
Rule: If a person buys cheese they will also buy apples
 - Confidence: 0.610
 - Support: 25

Rule #5
Rule: If a person buys apples they will also buy bananas
 - Confidence: 0.583
 - Support: 21
```

从排序结果来看，“顾客买苹果，也会买奶酪”和“顾客买奶酪，也会买香蕉”，这两条规则的支持度和置信度都很高。超市经理可以根据这些规则来调整商品摆放位置。例如，如果本周苹果促销，就在旁边摆上奶酪。但是香蕉和奶酪同时搞促销就没有多大意义了，因为我们发现购买奶酪的顾客中，接近66%的人即使不搞促销也会买香蕉——即使搞促销，也不会给销量带来多大提升。

从上面这个例子就能看出数据挖掘的洞察力有多强大。人们可以用数据挖掘技术探索数据集中各变量之间的关系，寻找新发现。接下来一节，我们看看数据挖掘的另一个功能：预测。

1.4　分类问题的简单示例

在上述亲和性分析例子中，我们寻找的是数据集中不同变量之间的相关性。而分类问题，只关注类别（也叫作目标）这个变量。上述例子中，假如我们关心的是怎样让顾客买更多的苹果，就可以把是否购买苹果作为类别，使用分类方法，只寻找促成顾客购买苹果的规则。

1.5　什么是分类

分类是数据挖掘领域最为常用的方法之一，不论是实际应用还是科研，都少不了它的身影。

对于分类问题，我们通常能拿到表示实际对象或事件的数据集，我们知道数据集中每一条数据所属的类别，这些类别把一条条数据划分为不同的类。什么是类别？类别的值又是怎么回事？我们来看下面几个例子。

- 根据检测数据确定植物的种类。类别的值为“植物属于哪个种类？”。
- 判断图像中有没有狗。类别是“图像里有狗吗？”。
- 根据化验结果，判断病人有没有患上癌症。类别是“病人得癌症了吗？”。

上述三个问题中有两个是二值（是/否）问题，但正如第一个确定植物类别的问题，多个类别的情况也很常见。

分类应用的目标是，根据已知类别的数据集，经过训练得到一个分类模型，再用模型对类别未知的数据进行分类。例如，我们可以对收到的邮件进行分类，标注哪些是自己希望收到的，哪些是垃圾邮件，然后用这些数据训练分类模型，实现一个垃圾邮件过滤器，这样以后再收到邮件，就不用自己去确认它是不是垃圾邮件了，过滤器就能帮你搞定。

1.5.1 准备数据集

我们接下来将使用著名的Iris植物分类数据集。这个数据集共有150条植物数据，每条数据都给出了四个特征：sepal length、sepal width、petal length、petal width（分别表示萼片和花瓣的长与宽），单位均为cm。这是数据挖掘中的经典数据集之一（1936年就用到了数据挖掘领域！）。该数据集共有三种类别：Iris Setosa（山鸢尾）、Iris Versicolour（变色鸢尾）和Iris Virginica（维吉尼亚鸢尾）。我们这里的分类目的是根据植物的特征推测它的种类。

scikit-learn库内置了该数据集，可直接导入。

```
from sklearn.datasets import load_iris
dataset = load_iris()
X = dataset.data
y = dataset.target
```

用print(dataset.DESCR)命令查看数据集，大概了解一下，包括特征的说明。

数据集中各特征值为连续型，也就是有无数个可能的值。测量得到的数据就是这个样子，比如，测量结果可能是1、1.2或1.25，等等。连续值的另一个特点是，如果两个值相近，表示相似度很大。一种萼片长1.2cm的植物跟一种萼片宽1.25cm的植物很相像。

与此相反，类别的取值为离散型。虽然常用数字表示类别，但是类别值不能根据数值大小比较相似性。Iris数据集用不同的数字表示不同的类别，比如类别0、1、2分别表示Iris Setosa、Iris Versicolour、Iris Virginica。但是这不能说明前两种植物，要比第一种和第三种更相近——尽管单看表示类别的数字时确实如此。在这里，数字表示类别，只能用来判断两种植物是否属于同一种

类别，而不能说明是否相似。

当然，还有其他类型的特征，后续章节会讲到其中几种。

数据集的特征为连续值，而我们即将使用的算法使用类别型特征值，因此我们需要把连续值转变为类别型，这个过程叫作离散化。

最简单的离散化算法，莫过于确定一个阈值，将低于该阈值的特征值置为0，高于阈值的置为1。我们把某项特征的阈值设定为该特征所有特征值的均值。每个特征的均值计算方法如下。

```
attribute_means = X.mean(axis=0)
```

我们得到了一个长度为4的数组，这正好是特征的数量。数组的第一项是第一个特征的均值，以此类推。接下来，用该方法将数据集打散，把连续的特征值转换为类别型。

```
X_d = np.array(X >= attribute_means, dtype='int')
```

后面的训练和测试，都将使用新得到的`x_d`数据集（打散后的数组X），而不再使用原来的数据集（X）。

1.5.2　实现 OneR 算法

OneR算法的思路很简单，它根据已有数据中，具有相同特征值的个体最可能属于哪个类别进行分类。OneR是One Rule（一条规则）的简写，表示我们只选取四个特征中分类效果最好的一个用作分类依据。后续章节中的分类算法比起OneR要复杂很多，但这个看似不起眼的简单算法，在很多真实数据集上表现得也不凡。

算法首先遍历每个特征的每一个取值，对于每一个特征值，统计它在各个类别中的出现次数，找到它出现次数最多的类别，并统计它在其他类别中的出现次数。

举例来说，假如数据集的某一个特征可以取0或1两个值。数据集共有三个类别。特征值为0的情况下，A类有20个这样的个体，B类有60个，C类也有20个。那么特征值为0的个体最可能属于B类，当然还有40个个体确实是特征值为0，但是它们不属于B类。将特征值为0的个体分到B类的错误率就是40%，因为有40个这样的个体分别属于A类和C类。特征值为1时，计算方法类似，不再赘述；其他各特征值最可能属于的类别及错误率的计算方法也一样。

统计完所有的特征值及其在每个类别的出现次数后，我们再来计算每个特征的错误率。计算方法为把它的各个取值的错误率相加，选取错误率最低的特征作为唯一的分类准则（OneR），用于接下来的分类。

现在，我们就来实现该算法。首先创建一个函数，根据待预测数据的某项特征值预测类别，并给出错误率。在这之前需要导入前面用过的`defaultdict`和`itemgetter`模块。

```
from collections import defaultdict
from operator import itemgetter
```

下面创建函数声明，参数分别是数据集、类别数组、选好的特征索引值、特征值。

```
def train_feature_value(X, y_true, feature_index, value):
```

接下来遍历数据集中每一条数据（代表一个个体），统计具有给定特征值的个体在各个类别中的出现次数。

```
    class_counts = defaultdict(int)
    for sample, y in zip(X, y_true):
        if sample[feature_index] == value:
            class_counts[y] += 1
```

对`class_counts`字典进行排序，找到最大值，就能找出具有给定特征值的个体在哪个类别中出现次数最多。

```
    sorted_class_counts = sorted(class_counts.items(),
      key=itemgetter(1), reverse=True)
        most_frequent_class = sorted_class_counts[0][0]
```

接着计算该条规则的错误率。OneR算法会把具有该项特征值的个体统统分到上面找到的出现次数最多的类别中。错误率为具有该特征值的个体在其他类别（除出现次数最多的类别之外的）中的出现次数，它表示的是分类规则不适用的个体的数量。

```
incorrect_predictions = [class_count for class_value, class_count
in class_counts.items()
if class_value != most_frequent_class]
error = sum(incorrect_predictions)
```

最后返回使用给定特征值得到的待预测个体的类别和错误率。

```
    return most_frequent_class, error
```

对于某项特征，遍历其每一个特征值，使用上述函数，就能得到预测结果和每个特征值所带来的错误率，然后把所有错误率累加起来，就能得到该特征的总错误率。我们来定义一个函数，实现这些操作。

函数声明如下，这次只用到三个参数，上面已经介绍过。

```
def train_on_feature(X, y_true, feature_index):
```

接下来找出给定特征共有几种不同的取值。下面这行代码`X[:,feature_index]`以数组的形式返回由`feature_index`所指的列。然后用`set`函数将数组转化为集合，从而找出有几种不同的取值。

```
    values = set(X[:,feature_index])
```

再创建字典predictors，用作预测器。字典的键为特征值，值为类别。比如键为1.5、值为2，表示特征值为1.5的个体属于类别2。创建errors列表，存储每个特征值的错误率。

```
predictors = {}
errors = []
```

函数的主干部分遍历选定特征的每个不同的特征值，用前面定义的train_feature_value()函数找出每个特征值最可能的类别，计算错误率，并将其分别保存到预测器predictors和errors中。

```
for current_value in values:
  most_frequent_class, error = train_feature_value(X,
    y_true, feature_index, current_value)
  predictors[current_value] = most_frequent_class
  errors.append(error)
```

最后，计算该规则的总错误率，返回预测器及总错误率。

```
total_error = sum(errors)
return predictors, total_error
```

1.5.3　测试算法

上一节中亲和性分析算法的目标是从数据集中发现用以指导实践的规则。而分类问题有所不同，我们想建立一个能够根据已有知识对没有见过的个体进行分类的模型。

我们因此把机器学习流程分为两步：训练和测试。在训练阶段，我们从数据集中取一部分数据，创建模型。在测试阶段，我们测试模型在数据集上的分类效果。考虑到模型的目标是对新个体进行分类，因此不能用测试数据训练模型，因为这样做容易导致过拟合问题。

过拟合指的是模型在训练集上表现很好，但对于没有见过的数据表现很差。解决方法很简单：千万不要用训练数据测试算法。详细的处理方法很复杂，后续章节会有所涉及；我们这里简单化处理，把数据集分为两个小部分，分别用于训练和测试。具体流程接下来会介绍。

scikit-learn库提供了一个将数据集切分为训练集和测试集的函数。

```
from sklearn.cross_validation import train_test_split
```

该函数根据设定的比例（默认把数据集的25%作为测试集）将数据集随机切分为两部分，以确保测试结果的可信度。

```
Xd_train, Xd_test, y_train, y_test = train_test_split(X_d, y, random_
state=14)
```

这样我们就得到了两个数据集：训练集Xd_train和测试集Xd_test。y_train和y_test分别为以上两个数据集的类别信息。

切分函数的第三个参数random_state用来指定切分的随机状态。每次切分，使用相同的随机状态，切分结果相同。虽然看起来是随机的，但是它所使用的算法是确定的，输出结果也是一致的。在书中所有用到random_state的地方，我建议你跟我使用同一个值，这样你得到的结果就应该跟我的相同，这样便于你验证结果。把random_state的值设置为none，每次切分结果将是真正随机的。

接下来，计算所有特征值的目标类别（预测器）。记得只使用训练集。遍历数据集中的每个特征，使用我们先前定义的函数train_on_feature()训练预测器，计算错误率。

```
all_predictors = {}
errors = {}
for feature_index in range(Xd_train.shape[1]):
  predictors, total_error = train_on_feature(Xd_train, y_train,
    feature_index)
  all_predictors[feature_index] = predictors
  errors[feature_index] = total_error
```

然后找出错误率最低的特征，作为分类的唯一规则。

```
best_feature, best_error = sorted(errors.items(), key=itemgetter(1))
[0]
```

对预测器进行排序，找到最佳特征值，创建model模型。

```
model = {'feature': best_feature,
  'predictor': all_predictors[best_feature][0]}
```

model模型是一个字典结构，包含两个元素：用于分类的特征和预测器。有了模型后，就可以根据特征值对没有见过的数据进行分类。示例如下：

```
variable = model['variable']
predictor = model['predictor']
prediction = predictor[int(sample[variable])]
```

我们经常需要一次对多条数据进行预测，为此用上面的代码实现了下面这个函数，通过遍历数据集中的每条数据来完成预测。

```
def predict(X_test, model):
    variable = model['variable']
    predictor = model['predictor']
    y_predicted = np.array([predictor[int(sample[variable])] for
      sample in X_test])
    return y_predicted
```

我们用上面这个函数预测测试集中每条数据的类别。

```
y_predicted = predict(X_test, model)
```

比较预测结果和实际类别，就能得到正确率是多少。

```
accuracy = np.mean(y_predicted == y_test) * 100
print("The test accuracy is {:.1f}%".format(accuracy))
```

输出结果为68%，对于只使用一条规则来说，这就很不错了！

1.6　小结

本章介绍了如何用Python进行数据挖掘。如果你能运行这一部分的代码[①]（见代码包第1章的文件夹），说明开发环境已搭建好，后续章节的大部分代码都能运行了。当然有些Python库还没装，随用随装就好。

我们用IPython Notebook运行了代码，好处是能及时看到一小块代码的输出。它功能强大，后面会继续使用。

我们举了一个简单的亲和性分析的例子，用它找出顾客经常一起购买的商品。这种探索性的分析方法用处很大，能帮助人们发现商业流程、某个环境或场景中的潜在规律。亲和性分析可用在商业、医疗、人工智能等领域，说不定能这些领域带来突破。

本章还通过OneR算法介绍了分类的应用。该算法寻找最佳的特征值用于分类，该特征值在训练集中哪个类别中出现的次数最多，待预测数据就属于哪个类别。

后续章节会扩展分类和亲和性分析的概念，同时还会介绍scikit-learn库以及它实现的一些数据挖掘算法。

① 如果.ipynb文件在Notebook中打开时报错，请用JSON检查工具查找有无不合法的JSON字符，自行调整一下。
——译者注

第2章 用scikit-learn估计器分类

用Python语言编写的scikit-learn库，实现了一系列数据挖掘算法，提供通用编程接口、标准化的测试和调参工具，便于用户尝试不同算法对其进行充分测试和查找最优参数值。有大量使用scikit-learn库的算法和工具。

本章讲解数据挖掘通用框架的搭建方法。有了这样一个框架，后续章节就可以把讲解重点放到数据挖掘应用和技术上面。

本章主要介绍如下几个概念。

- 估计器（Estimator）：用于分类、聚类和回归分析。
- 转换器（Transformer）：用于数据预处理和数据转换。
- 流水线（Pipeline）：组合数据挖掘流程，便于再次使用。

2.1 scikit-learn 估计器

为帮助用户实现大量分类算法，scikit-learn把相关功能封装成所谓的估计器。估计器用于分类任务，它主要包括以下两个函数。

- fit()：训练算法，设置内部参数。该函数接收训练集及其类别两个参数。
- predict()：参数为测试集。预测测试集类别，并返回一个包含测试集各条数据类别的数组。

大多数scikit-learn估计器接收和输出的数据格式均为numpy数组或类似格式。

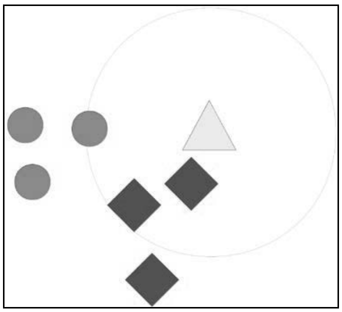
scikit-learn提供了大量估计器，其中有支持向量机（SVM）、随机森林、神经网络等，多数算法本书都会有所涉及。本章先介绍scikit-learn中的近邻算法。

> 我们需要安装matplotlib库，作图时会用到。最简单的安装方法就是用pip3来安装，在第1章安装scikit-learn时用过。
>
> ```
> $pip3 install matplotlib
> ```
>
> 安装过程中若遇到任何问题，请参考官方给出的安装指南：http://matplotlib.org/users/installing.html。

2.1.1　近邻算法

近邻算法可能是标准数据挖掘算法中最为直观的一种。为了对新个体进行分类，它查找训练集，找到与新个体最相似的那些个体，看看这些个体大多属于哪个类别，就把新个体分到哪个类别。

举例来说，我们要根据三角形更像什么（跟哪种图形离得更近），预测三角形的类别。我们找到三个离它最近的邻居：两个菱形和一个圆。菱形的数量多于圆，因此我们预测三角形的类别为菱形。



近邻算法几乎可以对任何数据集进行分类，但是，要计算数据集中每两个个体之间的距离，计算量很大。例如，数据集中个体数量为10时，需要计算45对不同个体之间的距离。然而，当个体数量为1000时，要计算大约50万对个体之间的距离！现在有很多提升该算法速度的方法，后续章节会讲到几种。

除了计算量大之外，该算法还有一个问题，就是在特征取离散值的数据集上表现很差。遇到这种情况，应该考虑使用其他算法。

2.1.2　距离度量

距离是数据挖掘的核心概念之一。我们往往需要知道两个个体之间的距离是多少。更进一步

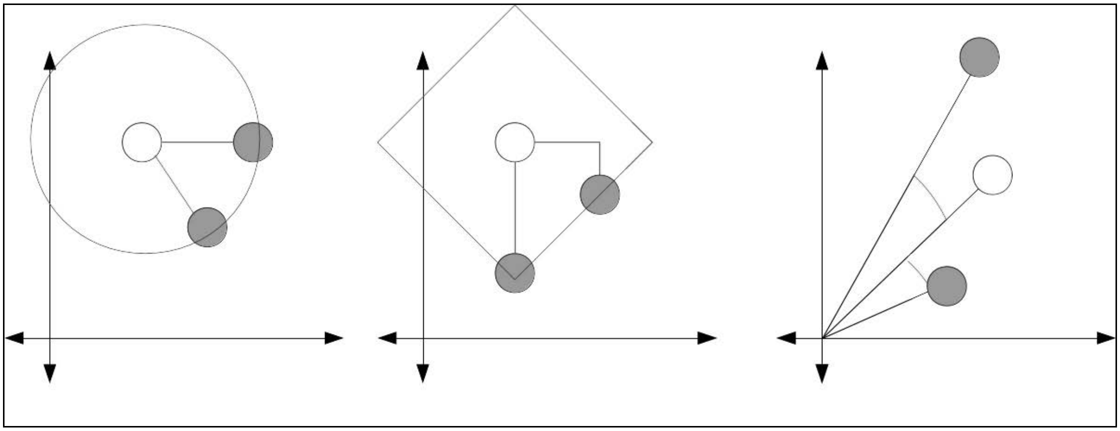
说，我们还得能够解决一对个体相对另一对个体是否更相近等问题。这类问题的解决方法，将直接影响分类结果。

人们耳熟能详的距离度量方法就是欧氏距离，即真实距离。假如你在图像中画两个点，用直尺测量这两个点之间的距离，得到的结果就是欧氏距离。稍微正式点来说，它其实是两个特征向量长度平方和的平方根。

欧氏距离确实很直观，但是如果某些特征比其他特征取值大很多，精确度就会比较差。此外，如果很多特征值为0，也就是所谓的稀疏矩阵，结果也不准确。这时可以用其他距离度量方法，常用的有曼哈顿距离和余弦距离。

曼哈顿距离为两个特征在标准坐标系中绝对轴距之和（没有使用平方距离）。拿国际象棋中的车[①]举例子，这样更形象。假如车每次只能走一格，那么它走到当前格子对角线那头，所走的距离就是曼哈顿距离。虽然异常值也会影响分类结果，但是其所受的影响要比欧氏距离小得多。

余弦距离更适合解决异常值和数据稀疏问题。直观上讲，余弦距离指的是特征向量夹角的余弦值。下面为以上三种距离的示意图。



在上面每张图中，两个灰圆与白圆之间的距离是相等的。左图是欧氏距离，因此两个灰圆都落在以白圆为圆心的同一个圆的圆周上，可以用尺子量量看。中间这幅图表示的是曼哈顿距离，它指的是从灰圆到白圆所走的横向和纵向距离之和，也叫街区距离（City Block），测量时要想象棋中车的走法。右图是余弦距离示意图，计算夹角之间的余弦值，忽略特征向量的长度。

采用哪种距离度量方法对最终结果有很大影响。例如，你的数据集有很多特征，但是如果任

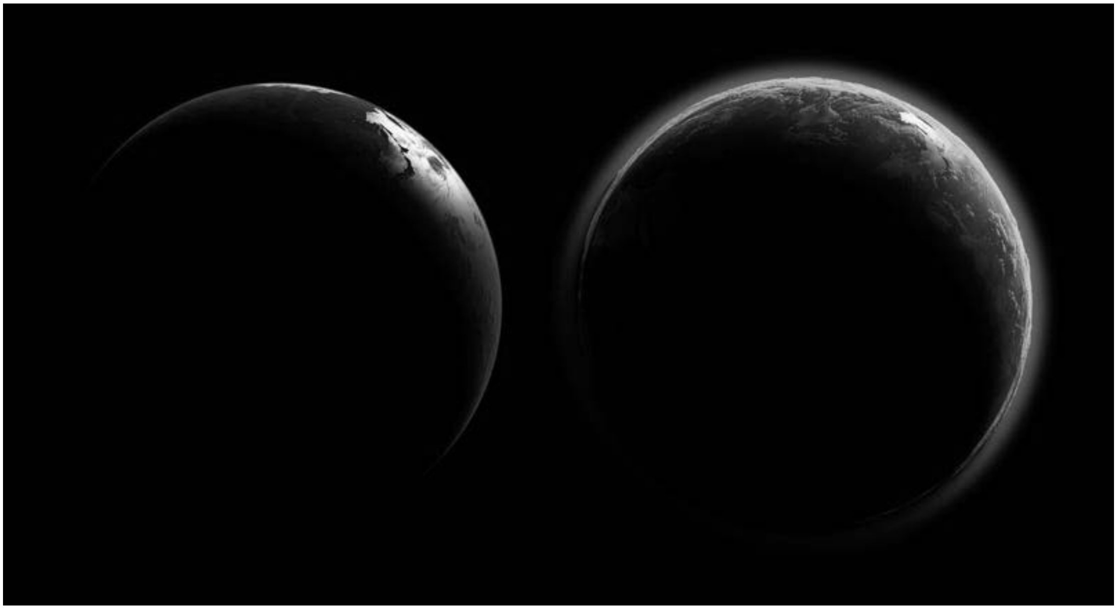
① 国际象棋中车的走法跟我国象棋中车的走法相同，只能沿水平或垂直方向移动。——译者注

意一对个体之间的欧氏距离都相等，那么你就没法通过欧氏距离进行比较了！曼哈顿距离在某些情况下具有更高的稳定性，但是如果数据集中某些特征值很大，用曼哈顿距离的话，这些特征会掩盖其他特征间的邻近关系。最后，再来说说余弦距离，它适用于特征向量很多的情况，但是它丢弃了向量长度所包含的在某些场景下可能会很有用的一些信息。

本章，我们主要介绍欧氏距离，其他距离后面章节再介绍。

2.1.3　加载数据集

即将用到的数据集叫作电离层（Ionosphere），这些数据是由高频天线收集的。这些天线的目的是侦测在电离层和高层大气中存不存在由自由电子组成的特殊结构。如果一条数据能给出特殊结构存在的证据，这条数据就属于好的那一类（在数据集中用“g”表示），否则就是坏的（用“b”表示）。我们要做的就是建立分类器，自动判断这些数据的好坏。



（图像来自https://www.flickr.com/photos/geckzilla/16149273389/）

Ionosphere数据集可以从UCI机器学习数据库下载，该数据库包含大量数据集，可用于多种数据挖掘任务。打开http://archive.ics.uci.edu/ml/datasets/Ionosphere，点击Data Folder。在随后打开的页面中，下载ionosphere.data和ionosphere.names文件。把这两个文件保存到用户主目录下的Data文件夹中。

> 主目录的位置取决于操作系统。Windows系统中通常为C:\Documents and Settings\username。Mac和Linux系统中为/home/username。如果你不确定，可以使用下面的Python代码输出主目录所在位置：
>
> ```
> import os
> print(os.path.expanduser("~"))
> ```

该数据集每行有35个值，前34个为17座天线采集的数据（每座天线采集两个数据）。最后一个值不是“g”就是“b”，表示数据的好坏，即是否提供了有价值的信息。

启动IPython Notebook服务器，新建名为Ionosphere Nearest Neighbors的笔记本文件。

首先，导入`numpy`和`csv`库，下面会用到。

```
import numpy as np
import csv
```

加载数据集前，用Data文件夹路径、数据集所在的文件夹名称和数据集名称组合成数据集文件的完整路径。

```
data_filename = os.path.join(data_folder, "Ionosphere",
"ionosphere.data")
```

创建Numpy数组`X`和`y`存放数据集。数据集大小已知，共有351行34列。在以后的实际工作中，如果你不知道数据集大小也没关系——后面章节会讲如何在不知道数据集大小的情况下加载它，具体怎么做现在不知道也没关系。

```
X = np.zeros((351, 34), dtype='float')
y = np.zeros((351,), dtype='bool')
```

Ionosphere数据集文件为CSV（Comma-Separated Values，用逗号分隔数据项）格式，这是常用的数据集存储格式。我们用`csv`模块来导入数据集文件，并创建`csv`阅读器对象。

```
with open(data_filename, 'r') as input_file:
    reader = csv.reader(input_file)
```

接着，遍历文件中的每一行数据。每行数据代表一组测量结果，我们可以将其称作数据集中的一个个体。用枚举函数来获得每行的索引号，在下面更新数据集`X`中的某一个体时会用到行号。

```
    for i, row in enumerate(reader):
```

获取每一个个体的前34个值，将其强制转化为浮点型，保存到`X`中。

```
        data = [float(datum) for datum in row[:-1]]
        X[i] = data
```

最后，获取每个个体最后一个表示类别的值，把字母转化为数字，如果类别为“g”，值为1，否则值为0。

```
    y[i] = row[-1] == 'g'
```

到此，我们就把数据集读到了数组x中，类别读入了数组y中，与我们在上一章的做法相同。

2.1.4　努力实现流程标准化

正如本章开头讲过的，scikit-learn估计器由两大函数组成：fit()和predict()。用fit方法在训练集上完成模型的创建，用predict方法在测试集上评估效果。

首先，需要创建训练集和测试集。导入并运行train_test_split函数。

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_
state=14)
```

然后，导入*K*近邻分类器这个类，并为其初始化一个实例。现阶段，参数用默认的即可，后面再讲参数调优。该算法默认选择5个近邻作为分类依据。

```
from sklearn.neighbors import KNeighborsClassifier
estimator = KNeighborsClassifier()
```

估计器创建好后，接下来就要用训练数据进行训练。*K*近邻估计器分析训练集中的数据，比较待分类的新数据点和训练集中的数据，找到新数据点的近邻。

```
estimator.fit(X_train, y_train)
```

接着，用测试集测试算法，评估它在测试集上的表现。

```
y_predicted = estimator.predict(X_test)
accuracy = np.mean(y_test == y_predicted) * 100
print("The accuracy is {0:.1f}%".format(accuracy))
```

正确率为86.4%。使用默认参数，只用少数几行代码就能达到这个效果，真是很厉害！虽然scikit-learn提供的大多数默认参数适用范围广，效果也不错，但我们还是要学着根据实验的实际情况，尽可能选用合适的参数值，争取达到最佳效果。

2.1.5　运行算法

在先前的几个实验中，我们把数据集分为训练集和测试集，用训练集训练算法，在测试集上评估效果。倘若碰巧走运，测试集很简单，我们就会觉得算法表现很出色。反之，我们可能会怀疑算法很糟糕。也许由于我们一时不走运，就把一个其实很不错的算法给无情抛弃了，这岂不是很可惜。

交叉检验能解决上述一次性测试所带来的问题。既然只切一次有问题，那就多切几次，多进行几次实验。每次切分时，都要保证这次得到的训练集和测试集与上次不一样，还要确保每条数

据都只能用来测试一次。算法描述如下。

(1) 将整个大数据集分为几个部分（fold[①]）。

(2) 对于每一部分执行以下操作：

- 将其中一部分作为当前测试集
- 用剩余部分训练算法
- 在当前测试集上测试算法

(3) 记录每次得分及平均得分。

(4) 在上述过程中，每条数据只能在测试集中出现一次，以减少（但不能完全规避）运气成分。

> 书中同一章的代码，前后是紧密联系的，所以需要把它们放到一个IPython Notebook笔记本文件中，除非我告诉你不必这么做。

scikit-learn提供了几种交叉检验方法。有个辅助函数实现了上述交叉检验步骤，现在把它导进来。

```
from sklearn.cross_validation import cross_val_score
```

> cross_val_score默认使用Stratified K Fold方法切分数据集，它大体上保证切分后得到的子数据集中类别分布相同，以避免某些子数据集出现类别分布失衡的情况。这个默认做法很不错，现阶段就不再把它搞复杂了。

我们就来试试这个函数吧，把完整的数据集和类别值传给它。

```
scores = cross_val_score(estimator, X, y, scoring='accuracy')
average_accuracy = np.mean(scores) * 100
print("The average accuracy is {0:.1f}%".format(average_accuracy))
```

哦，结果为82.3%，较之前稍微差点，但考虑到我们还没有尝试调整参数，这个结果还是相当不错的。下一节，我们就来研究怎么通过调整参数达到更理想的效果。

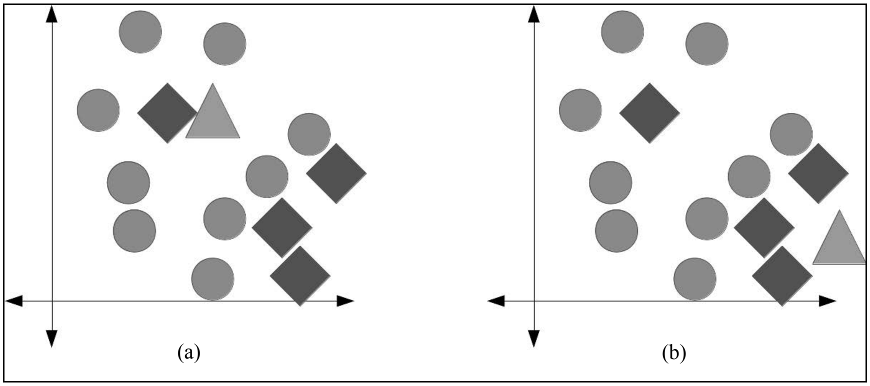
2.1.6 设置参数

几乎所有的数据挖掘算法都允许用户设置参数，这样做的好处是增强算法的泛化能力。但是，参数设置可是项技术活，选取好的参数值跟数据集的特征息息相关。

---

① 行话叫“折”。——译者注

近邻算法有多个参数，最重要的是选取多少个近邻作为预测依据。scikit-learn管这个参数叫n_neighbors。下图给出两个极端的例子，n_neighbors过小时，分类结果容易受干扰，随机性很强。相反，如果n_neighbors过大，实际近邻的影响将削弱。



左图(a)中，我们通常希望把测试数据（三角形）归到圆形类别。然而，如果n_neighbors的值为1，由于三角形附近红色菱形（很可能是噪音）的存在，导致分类结果为菱形，虽然菱形集中在右下角区域。右图(b)中，我们希望将测试数据归到菱形类别。然而，如果n_neighbors值为7，三个最近的邻居（都是菱形）被四个圆形给击败了，三角形也因此被归到圆形类别。

如果想测试一系列n_neighbors的值，比如从1到20，可以重复进行多次实验，观察不同的参数值所带来的结果之间的差异。

```
avg_scores = []
all_scores = []
parameter_values = list(range(1, 21))  # Include 20
for n_neighbors in parameter_values:
    estimator = KNeighborsClassifier(n_neighbors=n_neighbors)
    scores = cross_val_score(estimator, X, y, scoring='accuracy')
```

把不同n_neighbors值的得分和平均分保存起来，留作分析用。

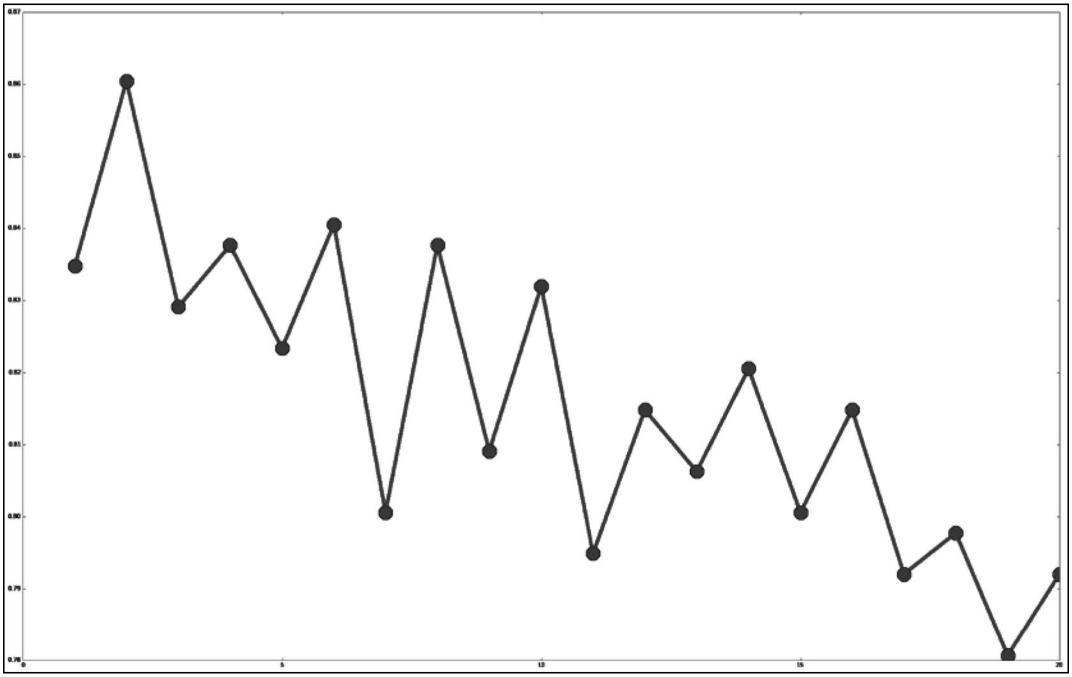
```
    avg_scores.append(np.mean(scores))
    all_scores.append(scores)
```

为了看起来更直观，我们可以用图表来表示n_neighbors的不同取值和分类正确率之间的关系。首先需要告诉IPython Notebook，我们要在笔记本中作图。

```
%matplotlib inline
```

然后，从matplotlib库导入pyplot，参数为近邻数和平均正确率。

```
from matplotlib import pyplot as plt plt.plot(parameter_values,
avg_scores, '-o')
```



从上图可以看到，虽然有很多曲折变化，但整体趋势是随着近邻数的增加，正确率不断下降。

2.2　流水线在预处理中的应用

现实中，物体不同特征的取值范围会非常广，它们的值域可能存在天壤之别。例如，测量动物的属性，会得到下面这样千差万别的特征值。

- 腿的数量：大多数动物有0到8条腿，但也有比这多得多的！
- 体重：从几微克到上百吨都有可能，有的蓝鲸重达190吨！
- 心脏数量：0到5之间，蚯蚓就有5颗心脏。

对于借助数学方法来比较特征的算法而言，它们很难理解特征在规模、范围和单位上的差异。如果我们在多种算法中使用上述特征，体重由于数值较大，可能都会是最显著的特征，但特征值大小实际上与该特征的分类效果没有任何关系。

不同特征的取值范围千差万别，常见的解决方法是对不同的特征进行规范化，使它们的特征值落在相同的值域或从属于某几个确定的类别，比如小、中和大。一旦解决这个问题，不同的特征类型对算法的影响将大大降低，分类正确率就能有大幅提升。

选择最具区分度的特征、创建新特征等都属于预处理的范畴。`scikit-learn`的预处理工具

叫作转换器（Transformer），它接受原始数据集，返回转换后的数据集。除了处理数值型特征，转换器还能用来抽取特征。在这里，我们只看下对数值型特征的预处理方法。

2.2.1　预处理示例

为了讲解需要，先来对Ionosphere数据集做些破坏。虽然这里的麻烦是人为制造的，但是这些问题在很多真实数据集里都存在。首先，为了不破坏原来的数据集，我们为其创建一个副本。

```
X_broken = np.array(X)
```

接下来，我们就要捣乱了，每隔一行，就把第二个特征的值除以10。

```
X_broken[:,::2] /= 10
```

理论上讲，这样做对结果影响应该不大。毕竟，除以10之后，各个特征相差不大。主要的问题是，数值范围变了，奇数行的第二个特征要比偶数行的大。再次计算正确率看一下效果。

```
estimator = KNeighborsClassifier()
original_scores = cross_val_score(estimator, X, y,
  scoring='accuracy')
print("The original average accuracy for is
{0:.1f}%".format(np.mean(original_scores) * 100))
broken_scores = cross_val_score(estimator, X_broken, y,
  scoring='accuracy')
print("The 'broken' average accuracy for is
  {0:.1f}%".format(np.mean(broken_scores) * 100))
```

还记得吧，在原始数据集上的正确率为82.3%，这次跌至71.5%。把特征值转变到0到1之间就能解决这个问题。

2.2.2　标准预处理

我们接下来用MinMaxScaler类进行基于特征的规范化。在本章的笔记本文件中，接着之前的代码写，首先导入所需的类。

```
from sklearn.preprocessing import MinMaxScaler
```

这个类可以把每个特征的值域规范化为0到1之间。最小值用0代替，最大值用1代替，其余值介于两者之间。

接下来，对数据集X进行预处理。我们在预处理器MinMaxScaler上调用转换函数。有些转换器要求像训练分类器那样先进行训练，但是MinMaxScaler不需要，直接调用fit_transform()函数，即可完成训练和转换。

```
X_transformed = MinMaxScaler().fit_transform(X)
```

X_transformed与X行列数相等，为同型矩阵。然而，前者每列值的值域为0到1。

还有很多其他类似的规范化方法，对于其他类型的应用和特征类型会很有用。

- 为使每条数据各特征值的和为1，使用sklearn.preprocessing.Normalizer。
- 为使各特征的均值为0，方差为1，使用sklearn.preprocessing.StandardScaler，常用作规范化的基准。
- 为将数值型特征的二值化，使用sklearn.preprocessing.Binarizer，大于阈值的为1，反之为0。

后续章节，将会组合运用上述预处理方法及其他转换器对象。

2.2.3　组装起来

现在我们把前几节所讲的代码组合起来，创建一套完整的工作流，处理被破坏过的数据集。

```
X_transformed = MinMaxScaler().fit_transform(X_broken)
estimator = KNeighborsClassifier()
transformed_scores = cross_val_score(estimator, X_transformed, y,
  scoring='accuracy')
print("The average accuracy for is
  {0:.1f}%".format(np.mean(transformed_scores) * 100))
```

正确率再次升到82.3%。MinMaxScaler将特征规范化到相同的值域，这样特征就不会仅仅因为值大而具备更强的区分度。简单总结下，异常值会影响近邻算法，不同算法对值域大小的敏感度不同。

2.3　流水线

随着实验的增加，操作的复杂程度也在提高。我们可能需要切分数据集，对特征进行二值化处理，以特征或数据集中的个体为基础规范化数据，除此之外还可能需要其他各种操作。

要跟踪记录所有这些操作可不容易，如果中间出点问题，先前实验的结果将很难再现。常见问题有落下步骤，数据转换错误，或进行了不必要的转换操作等。

另一个问题就是代码的先后顺序。上一节，我们创建了X_transformed数据集，然后创建了一个新的估计器用于交叉检验。如果有多个步骤，就需要跟踪代码中对数据集进行的每一步操作。

流水线结构就是用来解决这些问题的（当然不限于这些，下一章会讲到它在其他方面的应用）。流水线把这些步骤保存到数据挖掘的工作流中。之后你就可以用它们读入数据，做各种必要的预处理，然后给出预测结果。我们可以在cross_val_score等接收估计器的函数中使用流

水线。创建流水线前，先导入`Pipeline`对象。

```
from sklearn.pipeline import Pipeline
```

流水线的输入为一连串的数据挖掘步骤，其中最后一步必须是估计器，前几步是转换器。输入的数据集经过转换器的处理后，输出的结果作为下一步的输入。最后，用位于流水线最后一步的估计器对数据进行分类。我们流水线分为两大步。

(1) 用`MinMaxScaler`将特征取值范围规范到0~1。
(2) 指定`KNeighborsClassifier`分类器。

每一步都用元组（‘名称’，步骤）来表示。现在来创建流水线。

```
scaling_pipeline = Pipeline([('scale', MinMaxScaler()),
                             ('predict', KNeighborsClassifier())])
```

流水线的核心是元素为元组的列表。第一个元组规范特征取值范围，第二个元组实现预测功能。我们把第一步叫作规范特征取值（scale），第二步叫作预测（predict），也可以用其他名字。元组的第二部分是实际的转换器对象或估计器对象。

流水线写好后，运行它很简单。使用先前用到的交叉检验代码看一下实际效果。

```
scores = cross_val_score(scaling_pipeline, X_broken, y,
scoring='accuracy')
print("The pipeline scored an average accuracy for is {0:.1f}%".
format(np.mean(transformed_scores) * 100))
```

运行结果跟之前一样（82.3%），表明我们这次用到的步骤跟之前相同。

后续章节会使用更高级的测试方法，而设置流水线就很有必要，因为它能确保代码的复杂程度不至于超出掌控范围。

2.4　小结

本章，我们用`scikit-learn`库提供的几个方法，创建了运行和评估数据挖掘模型的标准工作流。还介绍了近邻算法，`scikit-learn`将其封装为一个估计器，使用起来很简单：首先调用`fit`函数在训练集上进行训练，然后用`predict`函数在测试集上评估效果。

本章还通过解决不同特征值域影响分类效果的问题，讲解了预处理方法，主要用到了转换器对象和`MinMaxScaler`类。它们也有训练（`fit`）和转换方法，在转换阶段，接收数据集，返回处理过的数据集。

下一章，我们尝试把学到的这些概念应用到一个大一点的例子上，学着预测NBA（美国职业篮球联赛）比赛结果，其中使用的数据集可是真实的哦。

第 3 章

用决策树预测获胜球队

本章介绍另一种分类算法——决策树，用它预测NBA篮球赛的获胜球队。比起其他算法，决策树有很多优点，其中最主要的一个优点是决策过程是机器和人都能看懂的，我们使用机器学习到的模型就能完成预测任务。正如我们将在本章讲到的，决策树的另一个优点则是它能处理多种不同类型的特征。

本章主要内容有：

- 用pandas库加载、处理数据
- 决策树
- 随机森林
- 对真实数据集进行数据挖掘
- 创建新特征，用强有力的框架对其进行测试

3.1 加载数据集

本章将介绍怎样预测NBA获胜球队。如果你看过NBA，可能知道比赛中两支球队比分咬得很紧，难分胜负，有时最后一分钟才能定输赢，因此预测赢家很难。很多体育赛事都有类似的特点，预期的大赢家也许当天被另一支队伍给打败了。

以往很多对体育赛事预测的研究表明，正确率因体育赛事而异，其上限在70%~80%之间。体育赛事预测多采用数据挖掘或统计学方法。

3.1.1 采集数据

我们将使用NBA 2013—2014赛季的比赛数据。http://Basketball-Reference.com网站提供了NBA及其他赛事的大量资料和统计数据。请按以下方法下载数据。

(1) 在浏览器中打开http://www.basketball-reference.com/leagues/NBA_2014_games.html。

(2) 点击标题Regular Season旁边的Export按钮。

(3) 将文件下载到Data文件夹，记录文件的路径。

数据文件格式为CSV，包含了NBA常规赛季的1230场比赛。

CSV为简单的文本格式文件，每行为一条用逗号分隔的数据（文件格式的名字就是这么来的）。在记事本里输入内容，保存时使用.csv扩展名，也能生成CSV文件。只要能阅读文本文件的编辑器，就能打开CSV文件，也可以用Excel把它作为电子表格打开。

我们用pandas（Python Data Analysis的简写，意为Python数据分析）库加载这些数据，pandas在数据处理方面特别有用。Python内置了读写CSV文件的`csv`库。但是，考虑到后面创建新特征时还要用到pandas更强大的一些函数，所以我们干脆用pandas加载数据文件。

> 本章需要安装pandas。最简单的方法就是用pip3来安装，第1章中安装scikit-learn库时用的就是pip3。pandas的安装方法如下：
>
> ```
> $pip3 install pandas
> ```
>
> 安装过程中若遇到任何困难，请访问http://pandas.pydata.org/getpandas.html，根据自己的系统，阅读相关安装指南。

3.1.2　用 pandas 加载数据集

pandas库是用来加载、管理和处理数据的。它在后台处理数据结构，支持诸计算均值等分析方法。

如果做过大量数据挖掘实验，就会发现自己翻来覆去地编写文件读取、特征抽取等函数。而这些函数每重新实现一次，都可能引入新错误。使用pandas等封装了很多功能的库，能有效减少反复实现上述函数所带来的工作量，并能保证代码的正确性。

本书后面会陆续介绍更多的数据挖掘案例，我们将大量使用pandas。

用`read_csv`函数就能加载数据集：

```
import pandas as pd
dataset = pd.read_csv(data_filename)
```

上述代码会加载数据集，将其保存到*数据框*（dataframe）中。数据框提供了一些非常好用的方法，后面会用到。我们来看看数据集是否有问题。输入以下代码，输出数据集的前5行：

```
dataset.ix[:5]
```

输出结果如下。

Out[47]:

Date	NaN	Visitor/Neutral	PTS	Home/Neutral	PTS	NaN	Notes
Tue Oct 29 2013	Box Score	Orlando Magic	87	Indiana Pacers	97	NaN	NaN
		Los Angeles Clippers	103	Los Angeles Lakers	116	NaN	NaN
		Chicago Bulls	95	Miami Heat	107	NaN	NaN
Wed Oct 30 2013	Box Score	Brooklyn Nets	94	Cleveland Cavaliers	98	NaN	NaN

从输出结果来看，这个数据集可以用，但存在几个小问题。下面我们就来修复这些问题。

3

3.1.3 数据集清洗

从上面的输出结果中，我们发现了以下几个问题。

- 日期是字符串格式，而不是日期对象。
- 第一行没有数据。
- 从视觉上检查结果，发现表头不完整或者不正确。

这些问题来自数据，我们可以改动数据本身，但是这样做的话，容易忘记之前做过哪些操作，落下步骤或是弄错哪一步，因而无法重现之前的结果。我们像前一章用流水线跟踪数据预处理流程那样，用pandas对原始数据进行预处理。

pandas.read_csv函数提供了可用来修复数据的参数，导入文件时指定这几个参数就好。导入后，我们还可以修改文件的头部，如下所示：

```
dataset = pd.read_csv(data_filename, parse_dates=["Date"],
  skiprows=[0,])
dataset.columns = ["Date", "Score Type", "Visitor Team",
  "VisitorPts", "Home Team", "HomePts", "OT?", "Notes"]
```

经过这些处理之后，结果会有很大改善，我们再来输出前5行看看：

```
dataset.ix[:5]
```

结果如下。

Out[48]:

	Date	Score Type	Visitor Team	VisitorPts	Home Team	HomePts	OT?	Notes
0	2013-10-29	Box Score	Orlando Magic	87	Indiana Pacers	97	NaN	NaN
1	2013-10-29	Box Score	Los Angeles Clippers	103	Los Angeles Lakers	116	NaN	NaN
2	2013-10-29	Box Score	Chicago Bulls	95	Miami Heat	107	NaN	NaN
3	2013-10-30	Box Score	Brooklyn Nets	94	Cleveland Cavaliers	98	NaN	NaN
4	2013-10-30	Box Score	Atlanta Hawks	109	Dallas Mavericks	118	NaN	NaN
5	2013-10-30	Box Score	Washington Wizards	102	Detroit Pistons	113	NaN	NaN

即使原始数据很规整，比如刚使用的这个，我们仍需要对其做些调整。其中一个原因是，文件可能来自不同的系统，由于存在兼容性问题，文件也许会发生变化。

既然数据已经准备好，在开始编写预测算法之前，我们先定下一个正确率作为基准。该基准任何算法都应该能达到。

每场比赛有两个队：主场队和客场队。最直接的方法就是拿几率作为基准，猜中的几率为50%。猜测任意一支球队获胜，都有一半胜算。

3.1.4　提取新特征

我们接下来通过组合和比较现有数据抽取特征。首先，确定类别值。在测试阶段，拿算法得到的分类结果与它对比，就能知道结果是否正确。类别可以有多种表示方法，我们这里用1表示主场队获胜，用0表示客场队获胜。对于篮球比赛而言，得分最多的队伍获胜。虽然数据集没有明确给出各球队的胜负情况，但是稍加计算就能得到。

找出主场获胜的球队：

```
dataset["HomeWin"] = dataset["VisitorPts"] < dataset["HomePts"]
```

我们把主场获胜球队的数据保存到NumPy数组里，稍后要用scikit-learn分类器对其进行处理。当前pandas和scikit-learn并没有进行整合，但是借助NumPy数组，它们配合地很好。我们用pandas抽取特征后再用scikit-learn抽取特征具体的值。

```
y_true = dataset["HomeWin"].values
```

上面的y_true数组保存的是类别数据，scikit-learn可直接读取该数组。

我们还可以创建一些特征用于数据挖掘。有时候，只要把原始数据丢给分类器就行了，但通常需要先抽取数值型或类别型特征。

首先，创建两个能帮助我们进行预测的特征，分别是这两支队伍上场比赛的胜负情况。赢得上场比赛，大致可以说明该球队水平较高。

遍历每一行数据，记录获胜球队。当到达一行新数据时，分别查看该行数据中的两支球队在各自的上一场比赛中有没有获胜的。

创建（默认）字典，存储球队上次比赛的结果。

```
from collections import defaultdict
won_last = defaultdict(int)
```

字典的键为球队，值为是否赢得上一场比赛。遍历所有行，在此过程中，更新每一行，为其增加两个特征值：两支球队在上场比赛有没有获胜。

```
for index, row in dataset.iterrows():
    home_team = row["Home Team"]
    visitor_team = row["Visitor Team"]
    row["HomeLastWin"] = won_last[home_team]
    row["VisitorLastWin"] = won_last[visitor_team]
    dataset.ix[index] = row
```

请注意，上述代码假定数据集是按照时间顺序排列的。我们所使用的数据集是按这种顺序排列的，如果你的数据集不是这样，你需要把代码中的`dataset.iterrows()`替换为`dataset.sort("Date").iterrows()`。

用当前比赛(遍历到的那一行数据所表示的比赛)的结果更新两支球队上场比赛的获胜情况，以便下次再遍历到这两支球队时使用。代码如下：

```
    won_last[home_team] = row["HomeWin"]
    won_last[visitor_team] = not row["HomeWin"]
```

上述代码运行结束后，我们多了两个新特征：HomeLastWin和VisitorLastWin。我们再来看下数据集。这次只看前5条意义不大。只有一个球队参加过两场比赛后，我们才知道它在上场比赛表现如何。以下代码将输出本赛季第20~25场比赛：

```
dataset.ix[20:25]
```

输出结果如下：

Out[52]:

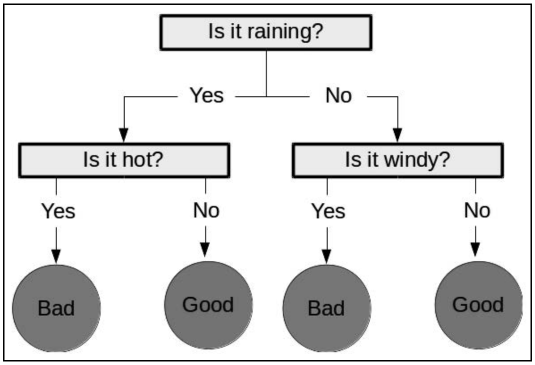
	Date	Score Type	Visitor Team	VisitorPts	Home Team	HomePts	OT?	Notes	HomeWin	HomeLastWin	VisitorLastWin
20	2013-11-01	Box Score	Milwaukee Bucks	105	Boston Celtics	98	NaN	NaN	False	False	False
21	2013-11-01	Box Score	Miami Heat	100	Brooklyn Nets	101	NaN	NaN	True	False	False
22	2013-11-01	Box Score	Cleveland Cavaliers	84	Charlotte Bobcats	90	NaN	NaN	True	False	True
23	2013-11-01	Box Score	Portland Trail Blazers	113	Denver Nuggets	98	NaN	NaN	False	False	False
24	2013-11-01	Box Score	Dallas Mavericks	105	Houston Rockets	113	NaN	NaN	True	True	True
25	2013-11-01	Box Score	San Antonio Spurs	91	Los Angeles Lakers	85	NaN	NaN	False	False	True

更换上述代码中的索引值，查看其他部分数据。别忘了，一共有1000多场比赛呢！

现在，每个队（包括上个赛季的冠军！）在数据集中第一次出现时，都假定它们在上场比赛中失败。其实可以用上一年的数据弥补缺失的信息，从而改进这个特征，但这里就先做简单处理了。

3.2 决策树

决策树是一种有监督的机器学习算法，它看起来就像是由一系列节点组成的流程图，其中位于上层节点的值决定下一步走向哪个节点。



跟大多数分类算法一样，决策树也分为两大步骤。

- ❑ 首先是训练阶段，用训练数据构造一棵树。上一章的近邻算法没有训练阶段，但是决策树需要。从这个意义上说，近邻算法是一种惰性算法，在用它进行分类时，它才开始干活。相反，决策树跟大多数机器学习方法类似，是一种积极学习的算法，在训练阶段完成模型的创建。
- ❑ 其次是预测阶段，用训练好的决策树预测新数据的类别。以上图为例，`["is raining", "very windy"]`的预测结果为“Bad”（坏天气）。

创建决策树的算法有多种，大都通过迭代生成一棵树。它们从根节点开始，选取最佳特征，用于第一个决策，到达下一个节点，选择下一个最佳特征，以此类推。当发现无法从增加树的层级中获得更多信息时，算法启动退出机制。

`scikit-learn`库实现了*分类回归树*（Classification and Regression Trees，CART）算法并将其作为生成决策树的默认算法，它支持连续型特征和类别型特征。

3.2.1 决策树中的参数

退出准则是决策树的一个重要特性。构建决策树时，最后几步决策仅依赖于少数个体，随意性大。使用特定节点作出推测容易导致过拟合训练数据，而使用退出准则可以防止决策精度过高。

除了设定退出准则外，也可以先创建一棵完整的树，再对其进行修剪，去掉对整个过程没有提供太多信息的节点。这个过程叫作剪枝（pruning）。

`scikit-learn`库实现的决策树算法给出了退出方法，使用下面这两个选项就可以达到目的。

- ❑ `min_samples_split`：指定创建一个新节点至少需要的个体数量。
- ❑ `min_samples_leaf`：指定为了保留节点，每个节点至少应该包含的个体数量。

第一个参数控制着决策节点的创建，第二个参数决定着决策节点能否被保留。

决策树的另一个参数是创建决策的标准，常用的有以下两个。

- 基尼不纯度（Gini impurity）：用于衡量决策节点错误预测新个体类别的比例。
- 信息增益（Information gain）：用信息论中的熵来表示决策节点提供多少新信息。

3.2.2 使用决策树

从scikit-learn库中导入DecisionTreeClassifier类，用它创建决策树。

```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(random_state=14)
```

> 我们再次设定random_state的值为14。本书中凡是用到random_state的地方，我们都用该值。使用相同的随机种子（random seed），能够保证几次实验结果相同。然而，在以后自己的实验中，为保证算法的性能不是与特定的随机状态值相关，在前后几次实验中，需使用不同的随机状态。

现在我们从pandas数据框中抽取数据，以使用scikit-learn分类器处理。指定需要的列，使用数据框的values属性，就能获取到每支球队的上一场比赛结果。

```
X_previouswins = dataset[["HomeLastWin", "VisitorLastWin"]].values
```

跟第2章的近邻算法类似，决策树也是一种估计器，因此它同样有fit和predict方法。我们仍然可以用cross_val_score方法来求得交叉检验的平均正确率：

```
scores = cross_val_score(clf, X_previouswins, y_true,
scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

正确率为56.1%，比起随机预测来要更准确！我们应该可以做得更好。从数据集中构建有效特征（Feature Engineering，特征工程）是数据挖掘的难点所在，好的特征直接关系到结果的正确率——甚至比选择合适的算法更重要！

3.3 NBA 比赛结果预测

尝试使用不同的特征，我们应该能做得更好。cross_val_score方法可用来测试模型的正确率。有了它，我们就可以尝试其他特征的分类效果。

好多潜在特征都可以拿来用。就我们这个挖掘任务而言，具体怎么选择特征呢？我们尝试问自己以下两个问题。

- ❑ 一般而言，什么样的球队水平更高？
- ❑ 两支球队上一次相遇时，谁是赢家？

我们还将加入新球队的数据，以检测算法是否能得到一个用来判断不同球队比赛情况的模型。

组装起来

对于上面第一个特征，我们创建一个叫作“主场队是否通常比对手水平高”的特征，并使用2013赛季的战绩作为特征取值来源。如果一支球队在2013赛季排名在对手前面，我们就认为它的水平更高。

战绩数据下载方法如下。

(1) 在浏览器中打开http://www.basketball-reference.com/leagues/NBA_2013_standings.html。
(2) 找到Expanded Standings部分，该部分包括所有球队的数据。
(3) 点击Export链接。
(4) 将数据保存到数据文件夹中。

回到IPython Notebook笔记本文件，在新格子里输入以下代码。请注意将战绩文件保存到`data_folder`变量指定的目录下。

```
standings_filename = os.path.join(data_folder,
"leagues_NBA_2013_standings_expanded-standings.csv")
standings = pd.read_csv(standings_filename, skiprows=[0,1])
```

在笔记本新格子里输入`standings`并运行，查看战绩。

```
standings
```

输出如下。

Out[57]:

	Rk	Team	Overall	Home	Road	E	W	A	C	SE	...	Post	≤3	≥10	Oct	Nov	Dec	Jan	Feb	Mar	Apr
0	1	Miami Heat	66-16	37-4	29-12	41-11	25-5	14-4	12-6	15-1	...	30-2	9-3	39-8	1-0	10-3	10-5	8-5	12-1	17-1	8-1
1	2	Oklahoma City Thunder	60-22	34-7	26-15	21-9	39-13	7-3	8-2	6-4	...	21-8	3-6	44-6	NaN	13-4	11-2	11-5	7-4	12-5	6-2
2	3	San Antonio Spurs	58-24	35-6	23-18	25-5	33-19	8-2	9-1	8-2	...	16-12	9-5	31-10	1-0	12-4	12-4	12-3	8-3	10-4	3-6
3	4	Denver Nuggets	57-25	38-3	19-22	19-11	38-14	5-5	10-0	4-6	...	24-4	11-7	28-8	0-1	8-8	9-6	12-3	8-4	13-2	7-1
4	5	Los Angeles Clippers	56-26	32-9	24-17	21-9	35-17	7-3	8-2	6-4	...	17-9	3-5	38-12	1-0	8-6	16-0	9-7	8-5	7-7	7-1
5	6	Memphis Grizzlies	56-26	32-9	24-17	22-8	34-18	8-2	8-2	6-4	...	23-8	6-4	28-9	0-1	12-1	7-7	10-7	9-2	11-6	7-2
6	7	New York Knicks	54-28	31-10	23-18	37-15	17-13	10-6	12-6	15-3	...	22-10	7-5	31-12	NaN	11-4	10-5	7-6	6-5	12-6	8-2
7	8	Brooklyn Nets	49-33	26-15	23-18	36-16	13-17	11-5	13-5	12-6	...	18-11	9-4	23-17	NaN	11-4	5-11	11-4	7-5	8-7	7-2
8	9	Indiana Pacers	49-32	30-11	19-21	31-20	18-12	6-11	13-3	12-6	...	17-11	4-9	27-14	1-0	7-8	10-5	9-6	9-3	11-5	2-5
9	10	Golden State Warriors	47-35	28-13	19-22	19-11	28-24	7-3	5-5	7-3	...	17-13	5-3	20-18	1-0	8-6	12-4	8-7	4-8	9-7	5-3

接下来，创建一个新特征，创建过程与上个特征类似。遍历每一行，查找主场队和客场队两

支球队的战绩。代码如下：

```
dataset["HomeTeamRanksHigher"] = 0
for index, row in dataset.iterrows():
    home_team = row["Home Team"]
    visitor_team = row["Visitor Team"]
```

有些球队2014赛季改名了，名字不同但其实还是同一支球队。类似情况在整合不同的数据集时经常遇到！所以在查找球队时，需要把它换成原来的名字，以确保正确找到该球队先前的排名。

3

```
    if home_team == "New Orleans Pelicans":
        home_team = "New Orleans Hornets"
    elif visitor_team == "New Orleans Pelicans":
        visitor_team = "New Orleans Hornets"
```

现在就能得到两支球队的排名，比较它们的排名，更新特征值。

```
    home_rank = standings[standings["Team"] ==
          home_team]["Rk"].values[0]
          visitor_rank = standings[standings["Team"] ==
          visitor_team]["Rk"].values[0]
          row["HomeTeamRanksHigher"] = int(home_rank > visitor_rank)
          dataset.ix[index] = row
```

接下来，用`cross_val_score`函数测试结果。首先，从数据集中抽取所需要的部分。

```
X_homehigher = dataset[["HomeLastWin", "VisitorLastWin",
  "HomeTeamRanksHigher"]].values
```

然后，创建`DecisionTreeClassifier`分类器，进行交叉检验，求得正确率。

```
clf = DecisionTreeClassifier(random_state=14)
scores = cross_val_score(clf, X_homehigher, y_true,
  scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

现在的正确率是60.3%——比我们之前的结果要好。还能再提高吗？

接下来，我们来统计两支球队上场比赛的情况，作为另一个特征。虽然球队排名有助于预测（排名靠前的胜算更大），但有时排名靠后的球队反而能战胜排名靠前的。原因有很多。例如，排名靠后的球队某些打法恰好能击中强者的软肋。该特征的创建方法与前一个特征类似，首先创建字典，保存上场比赛的获胜队伍，在数据框中建立新特征。代码如下：

```
last_match_winner = defaultdict(int)
dataset["HomeTeamWonLast"] = 0
```

然后，遍历每条数据，取到每场赛事的两支参赛队伍。

```
for index, row in dataset.iterrows():
    home_team = row["Home Team"]
```

```
    visitor_team = row["Visitor Team"]
```

不用考虑哪支球队是主场作战，我们想看一下这两支球队在上一场比赛中到底谁是赢家。因此，按照英文字母表顺序对球队名字进行排序，确保两支球队无论主客场作战，都使用相同的键。

```
    teams = tuple(sorted([home_team, visitor_team]))
```

通过查找字典，找到两支球队上次比赛的赢家。然后，更新数据框中这条数据。

```
    row["HomeTeamWonLast"] = 1 if last_match_winner[teams] ==
      row["Home Team"] else 0
      dataset.ix[index] = row
```

最后，更新last_match_winner字典，值为两支球队在当前场次比赛中的胜出者，两支球队再相逢时可将其作为参考。

```
    winner = row["Home Team"] if row["HomeWin"] else row
     ["Visitor Team"]
    last_match_winner[teams] = winner
```

下面，用新抽取的两个特征创建数据集。观察不同特征组合的分类效果。代码如下：

```
X_lastwinner = dataset[["HomeTeamRanksHigher", "HomeTeam
  WonLast"]].values
clf = DecisionTreeClassifier(random_state=14)
scores = cross_val_score(clf, X_lastwinner, y_true,
  scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

正确率为60.6%。结果越来越好了。

最后我们来看一下，决策树在训练数据量很大的情况下，能否得到有效的分类模型。我们将会为决策树添加球队，以检测它是否能整合新增的信息。

虽然决策树能够处理特征值为类别型的数据，但scikit-learn库所实现的决策树算法要求先对这类特征进行处理。用LabelEncoder转换器就能把字符串类型的球队名转化为整型。代码如下：

```
from sklearn.preprocessing import LabelEncoder
encoding = LabelEncoder()
```

将主场球队名称转化为整型：

```
encoding.fit(dataset["Home Team"].values)
```

接下来，抽取所有比赛的主客场球队的球队名（已转化为数值型）并将其组合（在NumPy中叫作“stacking”，是向量组合的意思）起来，形成一个矩阵。代码如下：

```
home_teams = encoding.transform(dataset["Home Team"].values)
visitor_teams = encoding.transform(dataset["Visitor Team"].values)
```

```
X_teams = np.vstack([home_teams, visitor_teams]).T
```

决策树可以用这些特征值进行训练，但DecisionTreeClassifier仍把它们当作连续型特征。例如，编号从0到16的17支球队，算法会认为球队1和2相似，而球队4和10不同。但其实这没意义，对于两支球队而言，它们要么是同一支球队，要么不同，没有中间状态！

为了消除这种和实际情况不一致的现象，我们可以使用OneHotEncoder转换器把这些整数转换为二进制数字。每个特征用一个二进制数字[①]来表示。例如，LabelEncoder为芝加哥公牛队分配的数值是7,那么OneHotEncoder为它分配的二进制数字的第七位就是1,其余队伍的第七位就是0。每个可能的特征值都这样处理，而数据集会变得很大。代码如下：

3

```
from sklearn.preprocessing import OneHotEncoder
onehot = OneHotEncoder()
```

在相同的数据集上进行预处理和训练操作，将结果保存起来备用。

```
X_teams_expanded = onehot.fit_transform(X_teams).todense()
```

接着，像之前那样在新数据集上调用决策树分类器。

```
clf = DecisionTreeClassifier(random_state=14)
scores = cross_val_score(clf, X_teams_expanded, y_true,
  scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

正确率为60%，比基准值要高，但是没有之前的效果好。原因可能在于特征数增加后，决策树处理不当。鉴于此，我们尝试修改算法，看看会不会起作用。数据挖掘有时就是不断尝试新算法、使用新特征这样一个过程。

3.4 随机森林

一棵决策树可以学到很复杂的规则。然而，很可能会导致过拟合问题——学到的规则只适用于训练集。解决方法之一就是调整决策树算法，限制它所学到的规则的数量。例如，把决策树的深度限制在三层，只让它学习从全局角度拆分数据集的最佳规则，不让它学习适用面很窄的特定规则，这些规则会将数据集进一步拆分为更加细致的群组。使用这种折中方案得到的决策树泛化能力强，但整体表现稍弱。

为了弥补上述方法的不足，我们可以创建多棵决策树，用它们分别进行预测，再根据少数服从多数的原则从多个预测结果中选择最终预测结果。这正是随机森林的工作原理。

但上述过程有两个问题。一是创建的多棵决策树在很大程度上是相同的——每次使用相同的

① 有多少个特征，二进制数字就有多少位。——译者注

输入，将得到相同的输出。我们只有一个训练集，如果尝试创建多棵决策树，它们的输入就可能相同（因此输出也相同）。解决方法是每次随机从数据集中选取一部分数据用作训练集。这个过程叫作装袋（bagging）。

第二个问题是用于前几个决策节点的特征非常突出。即使我们随机选取部分数据用作训练集，创建的决策树相似性仍旧很大。解决方法是，随机选取部分特征作为决策依据。

然后，使用随机从数据集中选取的数据和（几乎是）随机选取的特征，创建多棵决策树。这就是随机森林，虽然看上去不是那么直观，但这种算法在很多数据集上效果很好。

3.4.1　决策树的集成效果如何

随机森林算法内在的随机性让人感觉算法的好坏全靠运气。然而，通过对多棵几乎是随机创建的决策树的预测结果取均值，就能降低预测结果的不一致性。我们用方差来表示这种不一致。

*方差*是由训练集的变化引起的。决策树这类方差大的算法极易受到训练集变化的影响，从而产生过拟合问题。

> 对比来说，**偏误**（bias）是由算法中的假设引起的，而与数据集没有关系。比如，算法错误地假定所有特征呈正态分布，就会导致较高的误差。通过分析分类器的数据模型和实际数据集的匹配情况，就能降低偏误问题的负面影响。

对随机森林中大量决策树的预测结果取均值，能有效降低方差，这样得到的预测模型的总体正确率更高。

一般而言，决策树集成做出了如下假设：预测过程的误差具有随机性，且因分类器而异。因此，使用由多个模型得到的预测结果的均值，能够消除随机误差的影响——只保留正确的预测结果。本书中会介绍更多用集成方法消除误差的例子。

3.4.2　随机森林算法的参数

`scikit-learn`库中的`RandomForestClassifier`就是对随机森林算法的实现，它提供了一系列参数。因为它使用了`DecisionTreeClassifier`的大量实例，所以它俩的很多参数是一致的，比如决策标准（基尼不纯度/信息增益）、`max_features`和`min_samples_split`。

当然，集成过程还引入了一些新参数。

- `n_estimators`：用来指定创建决策树的数量。该值越高，所花时间越长，正确率（可能）也越高。

- oob_score：如果设置为真，测试时将不使用训练模型时用过的数据。
- n_jobs：采用并行计算方法训练决策树时所用到的内核数量。

scikit-learn库提供了用于并行计算的Joblib库。n_jobs指定所用的内核数。默认使用1个内核——如果CPU是多核的，可以多用几个，或者将其设置为–1，开动全部马力。

3.4.3 使用随机森林算法

scikit-learn库实现的随机森林算法使用估计器接口，用交叉检验方法调用它即可，代码跟之前大同小异。

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(random_state=14)
scores = cross_val_score(clf, X_teams, y_true, scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

只是靠更换分类器，正确率就提升了0.6%，达到60.6%。

随机森林使用不同的特征子集进行学习，应该比普通的决策树更为高效。下面来看一下多用几个特征效果如何。

```
X_all = np.hstack([X_home_higher, X_teams])
clf = RandomForestClassifier(random_state=14)
scores = cross_val_score(clf, X_all, y_true, scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

正确率为61.1%——又有所提升！可以使用GridSearchCV类搜索最佳参数，代码如下：

```
parameter_space = {
  "max_features": [2, 10, 'auto'],
  "n_estimators": [100,],
  "criterion": ["gini", "entropy"],
  "min_samples_leaf": [2, 4, 6],
}
clf = RandomForestClassifier(random_state=14)
grid = GridSearchCV(clf, parameter_space)
grid.fit(X_all, y_true)
print("Accuracy: {0:.1f}%".format(grid.best_score_ * 100))
```

这次正确率提升较大，达到了64.2%！

输出用网格搜索找到的最佳模型，查看都使用了哪些参数。代码如下：

```
print(grid.best_estimator_)
```

下面的代码将给出正确率最高的模型所用到的参数。

```
RandomForestClassifier(bootstrap=True, compute_importances=None,
  criterion='entropy', max_depth=None, max_features=2,
```

```
    max_leaf_nodes=None, min_density=None, min_samples_leaf=6,
      min_samples_split=2, n_estimators=100, n_jobs=1,
        oob_score=False, random_state=14, verbose=0)
```

3.4.4　创建新特征

从上述几个例子中，我们看到改变特征对算法的表现有很大影响。经过小规模测试发现，仅仅是因为选用不同的特征，正确率竟然提升了10%。

用pandas提供的函数创建特征。代码如下：

```
dataset["New Feature"] = feature_creator()
```

feature_creator函数返回数据集中每条数据的各个特征值。常用数据集作为参数。

```
dataset["New Feature"] = feature_creator(dataset)
```

最直接的做法是一开始为新特征设置默认的值，比如0。如下所示：

```
dataset["My New Feature"] = 0
```

接下来，遍历数据集，计算所需特征。本章曾多次用下面这种形式创建新特征。

```
  for index, row in dataset.iterrows():
    home_team = row["Home Team"]
    visitor_team = row["Visitor Team"]
    # Some calculation here to alter row
    dataset.ix[index] = row
```

请注意，上面这种遍历方法效率不高。如果你要用的话，请一次性处理所有特征。常用的“最佳做法”就是每条数据最好只处理一次。

你可以创建下述特征并看一下效果。

- 球队上次打比赛距今有多长时间？短期内连续作战，容易导致球员疲劳。
- 两支球队过去五场比赛结果如何？这两个数据要比HomeLastWin和VisitorLastWin更能反映球队的真实水平（抽取特征方法类似）。
- 球队是不是跟某支特定球队打比赛时发挥得更好？例如，球队在某个体育场里打比赛，即使是客场作战也能发挥得很好。

如果抽取上面这些特征遇到困难，请参考pandas的文档http://pandas.pydata.org/pandas-docs/stable/。此外，还可以尝试从Stack Overflow等社区寻求帮助。

更极致的做法是，借助球员数据来分析每个队的实力以预测输赢。其实，赌徒和体育博彩机构每天都在用这些复杂的特征预测赛事结果来牟利。

3.5　小结

本章使用`scikit-learn`库的另一个分类器`DecisionTreeClassifier`，并介绍了如何用pandas处理数据。我们分析了真实的NBA赛事的比赛结果数据，创建新特征用于分类，并在这个过程中发现即使是规整、干净的数据也可能存在一些小问题。

我们发现好的特征对提升正确率很有帮助，还使用了一种集成算法——随机森林，进一步提升正确率。

下一章将会扩展在第1章使用的亲和性分析算法，用来发现相似的电影。我们还将学到如何用算法解决排序问题，以及如何提升数据挖掘算法的可扩展性。

第 4 章

用亲和性分析方法推荐电影

本章学习如何用亲和性分析方法找出在什么情况下两个对象经常一起出现。通俗来讲，这也叫“购物篮分析”，因为曾有人用它找出哪些商品经常一起出售。

第3章关注的对象为球队，并用特征描述球队。本章所用到的电影评分数据有所不同，我们所关注的对象（电影）埋在数据中。本章数据挖掘任务的目标是找出对象同时出现的情况，也就是寻找用户同时喜欢几部电影的情况。

本章主要涉及以下几个概念。

- 亲和性分析
- 用Apriori算法挖掘关联特征
- 电影推荐
- 数据稀疏问题

4.1 亲和性分析

亲和性分析用来找出两个对象共同出现的情况。而前几章，我们关注的是同种对象之间的相似度。亲和性分析所用的数据通常为类似于交易信息的数据。从直观上来看，这些数据就像是商店的交易数据——从中能看出哪些商品是顾客一起购买的。

然而，亲和性分析方法的应用场景有很多，比如：

- 欺诈检测
- 顾客区分
- 软件优化
- 产品推荐

亲和性分析比分类更具探索性，因为通常我们无法拿到像在很多分类任务中所用的那样完整的数据集。例如，在电影推荐任务中，我们拿到的是不同用户对不同电影的评价。但是，每个用户不可能评价过所有电影，这就给亲和性分析带来一个不容忽视的大难题。如果用户没有评价过

一部电影，是因为他们不喜欢这部电影（据此就不推荐给他们），还是因为他们出于别的原因还没有评价？

本章不对上述问题做出解答，但是我们要思考数据集中类似这样的潜在问题该怎么解决。这些思考有助于提升推荐算法的准确性。

4.1.1 亲和性分析算法

我们在第1章介绍了一种基础的亲和性分析算法，尝试了所有可能的规则组合，计算了每条规则的置信度和支持度，并根据这两个标准进行排序，选取最佳规则。

然而，这个方法效率不高。好在第1章所使用的数据集中，每条交易数据只涉及五种商品。但现实并非如此，即使是再不起眼的小卖铺出售的商品也达上百种之多，网店更是有成千上万种商品（甚至几百万种！）。如果规则生成方法像第1章那样过于简单，计算这些规则所需要的时间复杂度将呈指数级增长。随着商品数量的增加，计算所有规则所需的时间增长得很快。更具体地说，所有可能的规则数量是2^n-1。数据集有5个特征，可能的规则就有31条；有10个特征，可能的规则就有1023条；仅仅有100个特征，规则数就能达到30位数字。即使计算能力大幅提升也未必能赶上在线商品的增长速度。因此，与其跟计算机过不去，不如寻找更加聪明的算法。

Apriori算法可以说是经典的亲和性分析算法。它只从数据集中频繁出现的商品中选取共同出现的商品组成频繁项集（frequent itemset），避免了上述复杂度呈指数级增长的问题。一旦找到频繁项集，生成关联规则就很容易了。

Apriori算法背后的原理简洁却不失巧妙。首先，确保了规则在数据集中有足够的支持度。Apriori算法的一个重要参数就是最小支持度。比如，要生成包含商品A、B的频繁项集（A, B），要求支持度至少为30，那么A和B都必须至少在数据集中出现30次。更大的频繁项集也要遵守该项约定，比如要生成频繁项集（A, B, C, D），那么子集(A, B, C)必须是频繁项集（当然D自己也要满足最小支持度标准）。

生成频繁项集后，将不再考虑其他可能的却不够频繁的项集（这样的集合有很多），从而大大减少测试新规则所需的时间。

其他亲和性分析算法有Eclat和频繁项集挖掘算法（FP-growth）。从数据挖掘角度看，这些算法比起基础的Apriori算法有很多改进，性能也有进一步提升。接下来，先来看一下最基础的Apriori算法。

4.1.2 选择参数

挖掘亲和性分析所用的关联规则之前，我们先用Apriori算法生成频繁项集。接着，通过检测

频繁项集中前提和结论的组合，生成关联规则（例如，如果用户喜欢电影X，那么他很可能喜欢电影Y）。

第一个阶段，需要为Apriori算法指定一个项集要成为频繁项集所需的最小支持度。任何小于最小支持度的项集将不再考虑。如果最小支持度值过小，Apriori算法要检测大量的项集，会拖慢的运行速度；最小支持度值过大的话，则只有很少的频繁项集。

找出频繁项集后，在第二个阶段，根据置信度选取关联规则。可以设定最小置信度，返回一部分规则，或者返回所有规则，让用户自己选。

本章，我们设定最小置信度，只返回高于它的规则。置信度过低将会导致规则支持度高，正确率低；置信度过高，导致正确率高，但是返回的规则少。

4.2　电影推荐问题

产品推荐技术是门大生意。网店经常用它向潜在用户推荐他们可能购买的产品。好的推荐算法能带来更高的销售业绩。每年有几百万乃至几千万用户进行网购，向他们推荐更多的商品，潜在收益着实可观。

产品推荐问题被人们研究了多年，但它一直不温不火，直到2007年到2009年间，Netflix公司推出数据建模大赛，并设立Netflix Prize奖项之后，才得到迅猛发展。该竞赛意在寻找比Netflix公司所使用的预测用户为电影打分的系统更准确的解决方案。最后获奖队伍以比现有系统高10个百分点的优势胜出。虽然这个改进看起来不是很大，但是Netflix公司却能借助它实现更精准的电影推荐服务，从而多赚上百万美元。

4.2.1　获取数据集

自打Netflix Prize奖项设立以来，美国明尼苏达大学的Grouplens研究团队公开了一系列用于测试推荐算法的数据集。其中，就包括几个大小不同的电影评分数据集，分别有10万、100万和1000万条电影评分数据。

数据集下载地址为http://grouplens.org/datasets/movielens/。本章将使用包含100万条数据的MovieLens数据集。下载数据集，解压到你的Data文件夹。启动IPython Notebook笔记本，输入以下代码。

```
import os
import pandas as pd
data_folder = os.path.join(os.path.expanduser("~"), "Data",
  "ml-100k")
ratings_filename = os.path.join(data_folder, "u.data")
```

确保`ratings_filename`指向解压后得到的文件夹中的`u.data`文件。

4.2.2　用 pandas 加载数据

MovieLens数据集非常规整，但是有几点跟`pandas.read_csv`方法的默认设置有出入，所以要调整参数设置。第一个问题是数据集每行的几个数据之间用制表符而不是逗号分隔。其次，没有表头，这表示数据集的第一行就是数据部分，我们需要手动为各列添加名称。

加载数据集时，把分隔符设置为制表符，告诉pandas不要把第一行作为表头（`header=None`），设置好各列的名称。代码如下：

```
all_ratings = pd.read_csv(ratings_filename, delimiter="\t",
    header=None, names = ["UserID", "MovieID", "Rating", "Datetime"])
```

虽然本章用不到，还是稍微提一下，你可以用下面的代码解析时间戳数据。

```
all_ratings["Datetime"] = pd.to_datetime(all_ratings['Datetime'],
  unit='s')
```

运行下面的代码，看一下前五条记录。

```
all_ratings[:5]
```

输出结果如下。

	UserID	MovieID	Rating	Datetime
0	196	242	3	1997-12-04 15:55:49
1	186	302	3	1998-04-04 19:22:22
2	22	377	1	1997-11-07 07:18:36
3	244	51	2	1997-11-27 05:02:03
4	166	346	1	1998-02-02 05:33:16

4.2.3　稀疏数据格式

这是一个稀疏数据集，我们可以将每一行想象成巨大特征矩阵的一个格子，前几章用到过这种矩阵。在这个矩阵中，每一行表示一个用户，每一列为一部电影。第一列为每一个用户给第一部电影打的分数，第二列为每一个用户给第二部电影打的分数，以此类推。

数据集中有1000名用户和1700部电影，这就意味着整个矩阵很大。将矩阵读到内存中及在它基础上进行计算可能存在难度。然而，这个矩阵的很多格子都是空的，也就是对大部分用户来说，他们只给少数几部电影打过分。比如用户#213没有为电影#675打过分，大部分用户没有为大部分电影打过分。

用上述图表中的格式也能表示矩阵，且更为紧凑。序号为0的那一行表示，用户#196在1997年12月4日为电影#242打了3分（满分是5分）。

任何没有出现在数据集中的用户和电影组合表示它们实际上是不存在的。这比起在内存中保存大量的0，节省了很多空间。这种格式叫作稀疏矩阵（sparse matrix）。根据经验来说，如果数据集中60%或以上的数据为0，就应该考虑使用稀疏矩阵，从而节省不少空间。

在对稀疏矩阵进行计算时，我们关注的通常不是那些不存在的数据，不会去比较众多的0值，相反我们关注的是现有数据，并对它们进行比较。

4.3　Apriori 算法的实现

本章数据挖掘的目标是生成如下形式的规则：如果用户喜欢某些电影，那么他们也会喜欢这部电影。作为对上述规则的扩展，我们还将讨论喜欢某几部电影的用户，是否喜欢另一部电影。

要解决以上问题，首先要确定用户是不是喜欢某一部电影。为此创建新特征`Favorable`，若用户喜欢该电影，值为`True`。

```
all_ratings["Favorable"] = all_ratings["Rating"] > 3
```

我们在数据集中看一下这个新特征。

```
all_ratings[10:15]
```

	UserID	MovieID	Rating	Datetime	Favorable
10	62	257	2	1997-11-12 22:07:14	False
11	286	1014	5	1997-11-17 15:38:45	True
12	200	222	5	1997-10-05 09:05:40	True
13	210	40	3	1998-03-27 21:59:54	False
14	224	29	3	1998-02-21 23:40:57	False

从数据集中选取一部分数据用作训练集，这能有效减少搜索空间，提升Apriori算法的速度。我们取前200名用户的打分数据。

```
ratings = all_ratings[all_ratings['UserID'].isin(range(200))]
```

接下来，新建一个数据集，只包括用户喜欢某部电影的数据行。

```
favorable_ratings = ratings[ratings["Favorable"]]
```

在生成项集时，需要搜索用户喜欢的电影。因此，接下来，我们需要知道每个用户各喜欢哪些电影，按照User ID进行分组，并遍历每个用户看过的每一部电影。

```
favorable_reviews_by_users = dict((k, frozenset(v.values))
                                   for k, v in favorable_ratings
                                   groupby("UserID")["MovieID"])
```

上面的代码把`v.values`存储为`frozenset`，便于快速判断用户是否为某部电影打过分。对

于这种操作，集合比列表速度快，在后面代码中还会用到。

最后，创建一个数据框，以便了解每部电影的影迷数量。

```
num_favorable_by_movie = ratings[["MovieID", "Favorable"]].
  groupby("MovieID").sum()
```

用以下代码查看最受欢迎的五部电影。

```
num_favorable_by_movie.sort("Favorable", ascending=False)[:5]
```

输出结果如下：

MovieID	Favorable
50	100
100	89
258	83
181	79
174	74

4.3.1 Apriori 算法

Apriori算法是亲和性分析的一部分，专门用于查找数据集中的频繁项集。基本流程是从前一步找到的频繁项集中找到新的备选集合，接着检测备选集合的频繁程度是否够高，然后算法像下面这样进行迭代。

(1) 把各项目放到只包含自己的项集中，生成最初的频繁项集。只使用达到最小支持度的项目。

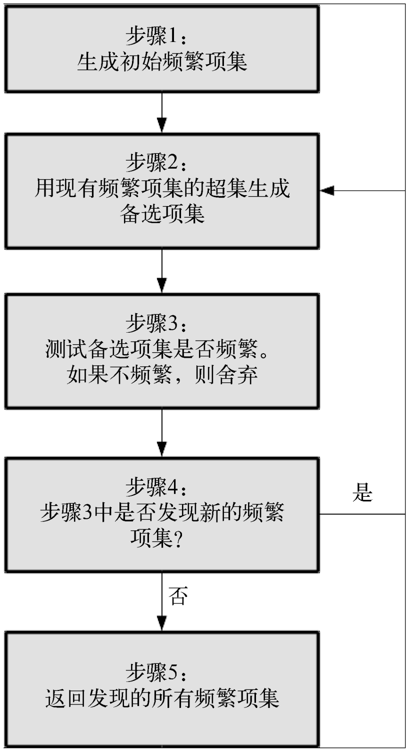
(2) 查找现有频繁项集的超集，发现新的频繁项集，并用其生成新的备选项集。

(3) 测试新生成的备选项集的频繁程度，如果不够频繁，则舍弃。如果没有新的频繁项集，就跳到最后一步。

(4) 存储新发现的频繁项集，跳到步骤(2)。

(5) 返回发现的所有频繁项集。

整个过程表示如下。



4.3.2　实现

Apriori算法第一次迭代时，新发现的项集长度为2，它们是步骤(1)中创建的项集的超集。第二次迭代（经过步骤(4)）中，新发现的项集长度为3。这有助于我们快速识别步骤(2)所需的项集。

我们把发现的频繁项集保存到以项集长度为键的字典中，便于根据长度查找，这样就可以找到最新发现的频繁项集。下面的代码初始化一个字典。

```
frequent_itemsets = {}
```

我们还需要确定项集要成为频繁项集所需的最小支持度。这个值需要根据数据集的具体情况来设定，可自行尝试其他值，建议每次只改动10个百分点，即使这样你可能也会发现算法运行时间变动很大！下面，设置最小支持度。

```
min_support = 50
```

我们先来实现Apriori算法的第一步，为每一部电影生成只包含它自己的项集，检测它是否够频繁。电影编号使用`frozenset`，后面要用到集合操作。此外，它们也可以用作字典的键（普通集合不可以）。代码如下：

```
frequent_itemsets[1] = dict((frozenset((movie_id,)),
                            row["Favorable"])
                            for movie_id, row in num_favorable_
                              by_movie.iterrows()
                            if row["Favorable"] > min_support)
```

接着，用一个函数来实现步骤(2)和(3)，它接收新发现的频繁项集，创建超集，检测频繁程度。下面为函数声明及字典初始化代码。

```
from collections import defaultdict
def find_frequent_itemsets(favorable_reviews_by_users, k_1_itemsets,
min_support):
    counts = defaultdict(int)
```


经验告诉我们，要尽量减少遍历数据的次数，所以每次调用函数时，再遍历数据。这样做效果不是很明显（因为数据集相对较小），但是数据集更大的情况下，就很有必要。我们来遍历所有用户和他们的打分数据。

```
    for user, reviews in favorable_reviews_by_users.items():
```

接着，遍历前面找出的项集，判断它们是否是当前评分项集的子集。如果是，表明用户已经为子集中的电影打过分。代码如下：

```
        for itemset in k_1_itemsets:
            if itemset.issubset(reviews):
```

接下来，遍历用户打过分却没有出现在项集里的电影，用它生成超集，更新该项集的计数。代码如下：

```
                for other_reviewed_movie in reviews - itemset:
                    current_superset = itemset | frozenset((other_
                                       reviewed_movie,))
                    counts[current_superset] += 1
```

函数最后检测达到支持度要求的项集，看它的频繁程度够不够，并返回其中的频繁项集。

```
    return dict([(itemset, frequency) for itemset, frequency in
counts.items() if frequency >= min_support])
```

创建循环，运行Apriori算法，存储算法运行过程中发现的新项集。循环体中，k表示即将发现的频繁项集的长度，用键k–1可以从`frequent_itemsets`字典中获取刚发现的频繁项集。新发现的频繁项集以长度为键，将其保存到字典中。代码如下：

```
for k in range(2, 20):
    cur_frequent_itemsets =
      find_frequent_itemsets(favorable_reviews_by_users,
        frequent_itemsets[k-1],
          min_support)
      frequent_itemsets[k] = cur_frequent_itemsets
```

如果在上述循环中没能找到任何新的频繁项集，就跳出循环（输出信息，告知我们没能找到长度为k的频繁项集）。

```
if len(cur_frequent_itemsets) == 0:
    print("Did not find any frequent itemsets of length {}".
      format(k))
    sys.stdout.flush()
    break
```

> 用sys.stdout.flush()方法，确保代码还在运行时，把缓冲区内容输出到终端。有时，在位于笔记本文件特定格子的大型循环中，代码结束运行前不会输出，用flush方法可以保证及时输出。但是，该方法不宜过多使用——flush操作（输出也是）所带来的计算开销会拖慢程序的运行速度。

如果确实找到了频繁项集，我们也让程序输出信息，告知我们它会再次运行。因为算法运行时间很长，所以每隔一段时间输出一下状态是很有必要的！代码如下：

```
else:
    print("I found {} frequent itemsets of length
    {}".format(len(cur_frequent_itemsets), k))
    sys.stdout.flush()
```

最后，循环结束，我们对只有一个元素的项集不再感兴趣——它们对生成关联规则没有用处——生成关联规则至少需要两个项目。删除长度为1的项集。代码如下：

```
del frequent_itemsets[1]
```

上面这些代码要好几分钟才能运行完，老机器需要的时间更长。如果你在本地运行代码有问题，可以考虑使用云主机提升速度，具体细节请见附录。

上述代码返回了不同长度的1718个频繁项集。你也许会发现随着项集长度的增加，项集数随着可用规则的增加而增长一段时间后才开始变少，减少是因为项集达不到最低支持度要求。项集的减少是Apriori算法的优点之一。如果我们搜索所有可能的项集（不只是频繁项集的超集），判断多余项集的频繁程度需要成千上万次查询。

4.4　抽取关联规则

Apriori算法结束后，我们得到了一系列频繁项集，这还不算是真正意义上的关联规则，但是很接近了。频繁项集是一组达到最小支持度的项目，而关联规则由前提和结论组成。

我们可以从频繁项集中抽取出关联规则，把其中几部电影作为前提，另一部电影作为结论组成如下形式的规则：*如果用户喜欢前提中的所有电影，那么他们也会喜欢结论中的电影。*

每一个项集都可用这种方式生成一条规则。

下面的代码通过遍历不同长度的频繁项集，为每个项集生成规则。

```
candidate_rules = []
for itemset_length, itemset_counts in frequent_itemsets.items():
    for itemset in itemset_counts.keys():
```

然后，遍历项集中的每一部电影，把它作为结论。项集中的其他电影作为前提，用前提和结论组成备选规则。

```
        for conclusion in itemset:
            premise = itemset - set((conclusion,))
            candidate_rules.append((premise, conclusion))
```


这样就能得到大量备选规则。通过以下代码查看前五条规则。

```
print(candidate_rules[:5])
```

输出结果如下：

```
[(frozenset({79}), 258), (frozenset({258}), 79), (frozenset({50}),
  64), (frozenset({64}), 50), (frozenset({127}), 181)]
```


在上述这些规则中，第一部分（frozenset）是作为规则前提的电影编号，后面的数字表示作为结论的电影编号。第一组数据表示如果用户喜欢电影79，他很可能喜欢电影258。

接下来，计算每条规则的置信度，计算方法跟第1章类似，只不过要根据这里新的数据格式做些改动。

我们需要先创建两个字典，用来存储规则应验（正例）和规则不适用（反例）的次数。代码如下：

```
correct_counts = defaultdict(int)
incorrect_counts = defaultdict(int)
```

遍历所有用户及其喜欢的电影数据，在这个过程中遍历每条关联规则。

```
for user, reviews in favorable_reviews_by_users.items():
    for candidate_rule in candidate_rules:
        premise, conclusion = candidate_rule
```

测试每条规则的前提对用户是否适用。换句话说，用户是否喜欢前提中的所有电影。代码如下：

```
        if premise.issubset(reviews):
```

如果前提符合，看一下用户是否喜欢结论中的电影。如果是的话，规则适用，反之，规则不适用。

```
        if premise.issubset(reviews):
```

```
            if conclusion in reviews:
                correct_counts[candidate_rule] += 1
            else:
                incorrect_counts[candidate_rule] += 1
```

用规则应验的次数除以前提条件出现的总次数，计算每条规则的置信度。

```
rule_confidence = {candidate_rule: correct_counts[candidate_rule]
  / float(correct_counts[candidate_rule] +
    incorrect_counts[candidate_rule])
              for candidate_rule in candidate_rules}
```

对置信度字典进行排序后，输出置信度最高的前五条规则。

```
from operator import itemgetter
sorted_confidence = sorted(rule_confidence.items(),
  key=itemgetter(1), reverse=True)
for index in range(5):
    print("Rule #{0}".format(index + 1))
    (premise, conclusion) = sorted_confidence[index][0]
    print("Rule: If a person recommends {0} they will also
      recommend {1}".format(premise, conclusion))
    print(" - Confidence:
      {0:.3f}".format(rule_confidence[(premise, conclusion)]))
    print("")
```

结果如下：

```
Rule #1
Rule: If a person recommends frozenset({64, 56, 98, 50, 7}) they will
also recommend 174
 - Confidence: 1.000

Rule #2
Rule: If a person recommends frozenset({98, 100, 172, 79, 50, 56})
they will also recommend 7
 - Confidence: 1.000

Rule #3
Rule: If a person recommends frozenset({98, 172, 181, 174, 7}) they
will also recommend 50
 - Confidence: 1.000

Rule #4
Rule: If a person recommends frozenset({64, 98, 100, 7, 172, 50}) they
will also recommend 174
 - Confidence: 1.000

Rule #5
Rule: If a person recommends frozenset({64, 1, 7, 172, 79, 50}) they
will also recommend 181
 - Confidence: 1.000
```

输出结果中只显示电影编号,而没有显示电影名字,很不友好。我们下载的数据集中的u.items文件里存储了电影名称和编号（还有体裁等信息）。

用pandas从u.items文件加载电影名称信息。关于该文件和类别的更多信息请见数据集中的README文件。u.items文件为CSV格式，但是用竖线分隔数据。读取时需要指定分隔符，设置表头和编码格式。每一列的名称是从README文件中找到的。

```
movie_name_filename = os.path.join(data_folder, "u.item")
movie_name_data = pd.read_csv(movie_name_filename, delimiter="|",
  header=None, encoding = "mac-roman")

movie_name_data.columns = ["MovieID", "Title", "Release Date",
  "Video Release", "IMDB", "<UNK>", "Action", "Adventure",
    "Animation", "Children's", "Comedy", "Crime", "Documentary",
      "Drama", "Fantasy", "Film-Noir",
  "Horror", "Musical", "Mystery", "Romance", "Sci-Fi", "Thriller",
    "War", "Western"]
```

4

既然电影名称对于理解数据很重要，我们就来创建一个用电影编号获取名称的函数，以免去每次都要人工查找的烦恼。函数声明如下：

```
def get_movie_name(movie_id):
```

在数据框`movie_name_data`中查找电影编号，找到后，只返回电影名称列的数据。

```
    title_object = movie_name_data[movie_name_data["MovieID"] ==
      movie_id]["Title"]
```

我们用`title_object`的`values`属性获取电影名称（不是存储在`title_object`中的`Series`对象）。我们只对第一个值感兴趣——当然每个电影编号只对应一个名称！

```
    title = title_object.values[0]
```

函数最后返回电影名称。

```
    return title
```

调整之前的代码，这样就能在输出的规则中显示电影名称。请在IPython Notebook笔记本文件的新格子里输入以下代码。

```
for index in range(5):
    print("Rule #{0}".format(index + 1))
    (premise, conclusion) = sorted_confidence[index][0]
    premise_names = ", ".join(get_movie_name(idx) for idx
      in premise)
    conclusion_name = get_movie_name(conclusion)
    print("Rule: If a person recommends {0} they will
      also recommend {1}".format(premise_names, conclusion_name))
    print(" - Confidence: {0:.3f}".format(confidence[(premise,
      conclusion)]))
print("")
```

结果清楚多了（还有些小问题，暂时先忽略）。

```
Rule #1
Rule: If a person recommends Shawshank Redemption, The (1994), Pulp
Fiction (1994), Silence of the Lambs, The (1991), Star Wars (1977),
Twelve Monkeys (1995) they will also recommend Raiders of the Lost Ark
(1981)
 - Confidence: 1.000

Rule #2
Rule: If a person recommends Silence of the Lambs, The (1991), Fargo
(1996), Empire Strikes Back, The (1980), Fugitive, The (1993), Star
Wars (1977), Pulp Fiction (1994) they will also recommend Twelve
Monkeys (1995)
 - Confidence: 1.000

Rule #3
Rule: If a person recommends Silence of the Lambs, The (1991), Empire
Strikes Back, The (1980), Return of the Jedi (1983), Raiders of the
Lost Ark (1981), Twelve Monkeys (1995) they will also recommend Star
Wars (1977)
 - Confidence: 1.000

Rule #4
Rule: If a person recommends Shawshank Redemption, The (1994), Silence
of the Lambs, The (1991), Fargo (1996), Twelve Monkeys (1995), Empire
Strikes Back, The (1980), Star Wars (1977) they will also recommend
Raiders of the Lost Ark (1981)
 - Confidence: 1.000

Rule #5
Rule: If a person recommends Shawshank Redemption, The (1994), Toy
Story (1995), Twelve Monkeys (1995), Empire Strikes Back, The (1980),
Fugitive, The (1993), Star Wars (1977) they will also recommend Return
of the Jedi (1983)
 - Confidence: 1.000
```

评估

广义上讲，我们可以拿评估分类算法的那一套来用。训练前，留出一部分数据用于测试，评估发现的规则在测试集上的表现。

如果这样做的话，我们就需要计算每条规则在测试集中的置信度。

这里我们不打算用正式的评价指标，只是简单看一下每条规则的表现，寻找好的规则。

首先，抽取所有没有用于训练的数据作为测试集。训练集用前200名用户的打分数据，测试集就用剩下的数据。就训练集来说，我们会为该数据集中的每一位用户获取最喜欢的电影。代码如下：

```
test_dataset =
all_ratings[~all_ratings['UserID'].isin(range(200))]
test_favorable = test_dataset[test_dataset["Favorable"]]
test_favorable_by_users = dict((k, frozenset(v.values)) for k, v
  in test_favorable.groupby("UserID")["MovieID"])
```

接着，计算规则应验的数量，方法跟之前相同。唯一的不同就是这次使用的是测试数据而不是训练数据。代码如下：

```
correct_counts = defaultdict(int)
incorrect_counts = defaultdict(int)
for user, reviews in test_favorable_by_users.items():
    for candidate_rule in candidate_rules:
        premise, conclusion = candidate_rule
        if premise.issubset(reviews):
            if conclusion in reviews:
                correct_counts[candidate_rule] += 1
            else:
                incorrect_counts[candidate_rule] += 1
```

接下来，计算所有应验规则的置信度。

```
test_confidence = {candidate_rule: correct_counts[candidate_rule]
  / float(correct_counts[candidate_rule] + incorrect_counts
    [candidate_rule])
  for candidate_rule in rule_confidence}
```

最后，输出用电影名称而不是电影编号表示的最佳关联规则。

```
for index in range(5):
    print("Rule #{0}".format(index + 1))
    (premise, conclusion) = sorted_confidence[index][0]
    premise_names = ", ".join(get_movie_name(idx) for idx in
      premise)
    conclusion_name = get_movie_name(conclusion)
    print("Rule: If a person recommends {0} they will also
      recommend {1}".format(premise_names, conclusion_name))
    print(" - Train Confidence:
      {0:.3f}".format(rule_confidence.get((premise, conclusion),
        -1)))
    print(" - Test Confidence:
     {0:.3f}".format(test_confidence.get((premise, conclusion),
        -1)))
     print("")
```

我们来看一下在新数据集上哪些规则最适用。

```
Rule #1
Rule: If a person recommends Shawshank Redemption, The (1994), Pulp
Fiction (1994), Silence of the Lambs, The (1991), Star Wars (1977),
Twelve Monkeys (1995) they will also recommend Raiders of the Lost Ark
(1981)
 - Train Confidence: 1.000
```

```
 - Test Confidence: 0.909

Rule #2
Rule: If a person recommends Silence of the Lambs, The (1991), Fargo
(1996), Empire Strikes Back, The (1980), Fugitive, The (1993), Star
Wars (1977), Pulp Fiction (1994) they will also recommend Twelve
Monkeys (1995)
 - Train Confidence: 1.000
 - Test Confidence: 0.609

Rule #3
Rule: If a person recommends Silence of the Lambs, The (1991), Empire
Strikes Back, The (1980), Return of the Jedi (1983), Raiders of the
Lost Ark (1981), Twelve Monkeys (1995) they will also recommend Star
Wars (1977)
 - Train Confidence: 1.000
 - Test Confidence: 0.946

Rule #4
Rule: If a person recommends Shawshank Redemption, The (1994), Silence
of the Lambs, The (1991), Fargo (1996), Twelve Monkeys (1995), Empire
Strikes Back, The (1980), Star Wars (1977) they will also recommend
Raiders of the Lost Ark (1981)
 - Train Confidence: 1.000
 - Test Confidence: 0.971

Rule #5
Rule: If a person recommends Shawshank Redemption, The (1994), Toy
Story (1995), Twelve Monkeys (1995), Empire Strikes Back, The (1980),
Fugitive, The (1993), Star Wars (1977) they will also recommend Return
of the Jedi (1983)
 - Train Confidence: 1.000
 - Test Confidence: 0.900
```

举例来说，第二条规则，在训练集中置信度为1，但在测试集上正确率只有60%。但前十条规则中，其他几条规则在测试集上置信度也很高，用它们来推荐电影效果不错。

> 浏览剩余规则，你可能发现有些规则的置信度为–1。置信度一般为0到1之间的值。负数表示这条规则在测试集中根本不存在。

4.5　小结

本章把亲和性分析用到电影推荐上，从大量电影打分数据中找到可用于电影推荐的关联规则。整个过程分为两大阶段。首先，借助Apriori算法寻找数据中的频繁项集。然后，根据找到的频繁项集，生成关联规则。

由于数据集较大，Apriori算法就很有必要。虽然第1章使用最笨的方法来寻找规则，但是在

这里就不适用了，由于时间复杂度呈指数级增长，我们需要寻找更巧妙的解决方案。这种现象在数据挖掘中很常见：虽然可以用最笨的方法穷尽所有情况来解决问题，但在数据量很大的情况下必须要使用更灵活、更快捷的算法。

我们用一部分数据作为训练集以发现关联规则，在剩余数据——测试集上进行测试。可以用前几章的交叉检验方法对每条规则的效果进行更为充分的评估。

到目前为止，我们使用的数据集都有很明显的特征。然而，不是所有数据集都是这个样子。下一章将学习用`scikit-learn`的转换器（第3章曾介绍过）从数据中抽取特征。还将讨论怎样实现自己的转换器和扩展已有转换器，并介绍相关概念。

第 5 章 用转换器抽取特征

前几章所用到的数据集都是用特征来描述的。虽然上一章的数据集为交易类型的数据，形式稍有不同，但归根结底还是一种用特征来描述的数据集。

除此之外，还有很多其他类型的数据集，比如文本、图像、声音、视频甚至是真实的物体。然而，大多数数据挖掘算法都依赖于数值或类别型特征。这表明在使用数据挖掘算法处理它们之前，需要找到一种表示它们的方法。

本章所讨论的是如何从数据集中抽取数值和类别型特征，并选出最佳特征，前提是数据集确实包含这些特征。我们还会介绍特征抽取的常用模式和技巧。

本章主要介绍以下几个概念。

- 从数据集中抽取特征
- 创建新特征
- 选取好特征
- 创建转换器，处理数据集

5.1 特征抽取

特征抽取是数据挖掘任务最为重要的一个环节，一般而言，它对最终结果的影响要高过数据挖掘算法本身。不幸的是，关于怎样选取好的特征，还没有严格、快捷的规则可循，其实这也是数据挖掘科学更像是一门艺术的所在。创建好的规则离不开直觉，还需要专业领域知识和数据挖掘经验，光有这些还不够，还得不停地尝试、摸索，在试错中前进，有时多少还要靠点运气。

5.1.1 在模型中表示事实

不是所有的数据集都是用特征来表示的。数据集可以是一位作家所写的全部书籍，也可以是1979年以来所有电影的胶片，还可以是馆藏的一批历史文物。

我们可能想对以上数据集做数据挖掘。对于作家的作品，我们想知道这位作家都写过哪些主题；对于电影，我们想知道女性形象是怎么塑造的；对于文物，我们想知道它们是何方产物。我们没办法把实物直接塞进决策树来得到结果。

只有先把现实用特征表示出来，才能借助数据挖掘的力量找到问题的答案。特征可用于建模，模型以机器挖掘算法能够理解的近似的方式来表示现实。因此可以说，模型描述了客观世界中对象的某些方面，是它的一个简化版本。举例来说，象棋就是对过往战事简化后得到的一种模型。

特征选择的另一个优点在于：降低真实世界的复杂度，模型比现实更容易操纵。想象一下，向一个没有任何背景知识的人介绍一种新事物，要给出恰如其分、精确又不失全面的描述需要提供多少信息。我们可能需要介绍其尺寸、重量、质地、成分、年龄、瑕疵、用途、产地等信息。

实物的复杂性对目前的算法而言过于复杂，我们退而求其次，使用更为简洁的模型来表示实物。

简化要以数据挖掘应用的目标为核心。本书后面会讲到聚类，对这类应用而言，特征选取至关重要，如果随意选取特征，分簇结果因选择的特征而异，呈现出很强的随机性。

降低复杂性有好处，但也有不足，简化会忽略很多细节，甚至会抛弃很多对数据挖掘算法能起到帮助作用的信息。

我们应该始终关注怎么用模型来表示现实，要多考虑数据挖掘的目标，而不是轻率地用我们过去用过的特征。要经常想想我们的目标是什么。第3章创建特征时充分考虑目标（预测赢家），使用篮球比赛领域相关知识，找出哪些因素与比赛结果密切相关，据此再创建新特征。

> 不是所有的特征必须是数值或类别型值，直接用于文本、图像和其他数据结构的算法已经研究出来了。不幸的是，篇幅有限，本书主要讲解数值和类别型特征，不涉及上述算法。

用Adult数据集讲解如何借助特征为复杂的现实世界建模再好不过。我们这里数据挖掘的目标是，预测一个人是否年收入多于五万美元。数据集下载地址为http://archive.ics.uci.edu/ml/datasets/Adult，点击Data Folder链接。下载adult.data和adult.names文件，在数据集文件夹（主目录下Data文件夹）中创建Adult文件夹，将这两个文件放到里面。

数据集用特征描述了一个个活生生的人及其他们所处的环境、背景和生活状况。

打开IPython Notebook，新建一个笔记本文件用于本章实验，导入pandas模块，指定数据集文件的路径，用pandas加载数据集文件。

```
import os
import pandas as pd
data_folder = os.path.join(os.path.expanduser("~"), "Data",
```

```
  "Adult")
adult_filename = os.path.join(data_folder, "adult.data")
Using pandas as before, we load the file with read_csv:
adult = pd.read_csv(adult_filename, header=None,
    names=["Age", "Work-Class", "fnlwgt",
    "Education", "Education-Num",
    "Marital-Status", "Occupation",
    "Relationship", "Race", "Sex",
    "Capital-gain", "Capital-loss",
    "Hours-per-week", "Native-Country",
    "Earnings-Raw"])
```

大部分代码在上一章都见过。

adult.data文件末尾有两处空行。pandas默认把倒数第二行空行作为一条有效数据读入（只不过各列的值为空）。我们需要删除包含无效数字的行（设置`inplace`参数为真，表示改动当前数据框，而不是新建一个）。

```
adult.dropna(how='all', inplace=True)
```

输入下面代码并运行，查看所有的特征名称。

```
adult.columns
```

下面为所有保存在pandas的`Index`对象中的特征名称。

```
Index(['Age', 'Work-Class', 'fnlwgt', 'Education',
'Education-Num', 'Marital-Status', 'Occupation', 'Relationship',
'Race', 'Sex', 'Capital-gain', 'Capital-loss', 'Hours-per-week',
'Native-Country', 'Earnings-Raw'], dtype='object')
```

5.1.2　通用的特征创建模式

纵然有数不清的特征创建方法，但其中有一些是适用于不同学科的通用模式。然而，选择合适的特征是项技术活，需要考虑特征和最终结果之间的相互关系。正如谚语所说的，不要用封面来评价书的好坏——如果你对书的内容感兴趣，书的大小可能不用考虑。

一些通用特征关注的是所研究对象的物理属性。

- 空间属性，比如对象的长、宽、高
- 重量和（或）密度
- 年龄、使用年限或成分
- 种类
- 品质

另一些特征可能与对象的使用或历史相关。

- 生产商、出版商或创造者
- 生产时间
- 使用方法

还有一些特征从组成成分角度描述对象。

- 次级成分的频率，如一本书中某个单词的词频
- 次级成分的数量和（或）次级成分种类的数量
- 次级成分的平均大小，例如平均句长

序数特征可对其排序、分组。正如前几章所见到的，特征可以是数值或类别型特征。数值特征通常被看作是有顺序的（ordinal）。例如，爱丽丝、鲍勃、查理三个人身高分别为1.5m、1.6m和1.7m。我们可以说爱丽丝和鲍勃，比起爱丽丝和查理，在身高上更相似。

上一节我们导入的数据集，包含连续特征、序数特征。例如，Hours-per-week特征表示一个人每周的工作时间。这个特征可用来计算平均工作时间、标准差、最大值和最小值。pandas提供了常见统计量的计算方法。

5

```
adult["Hours-per-week"].describe()
```

输出结果让我们对这个特征有了更多了解。

```
count    32561.000000
mean        40.437456
std         12.347429
min          1.000000
25%         40.000000
50%         40.000000
75%         45.000000
max         99.000000
dtype: float64
```

这些统计方法对其他特征可能没有意义。例如，计算受教育程度的和就讲不通。

还有些特征，它们虽然不是数值，但仍然属于序数值，比如Adult数据集中的Education（教育）特征，学士学位比高中毕业，在受教育程度上，要高一个层次，而高中毕业又比没有完成高中学业高一个层次。计算它俩的均值就没有意义，但是我们可以找到一种近似的表示方法。数据集有个叫作Education-Num（受教育年限）的特征，它的取值基本上就是每个教育阶段的年限。这样我们就能快速计算受教育年限的均值。

```
adult["Education-Num"].median()
```

结果为10，或者可以表述为刚好读完高一。如果没有受教育年限数据，我们为不同教育阶段指定数字编号，也可以计算均值。

特征也可以是类别型的。例如，一个球可以是网球、棒球、足球或其他种类。类别型特征也

可以称为名义特征（nominal）。对于名义特征而言，几个值之间要么相同要么不同。球类可以根据大小或重量排序，但是无法仅根据类别值进行量上的比较。网球不是棒球，也不是足球。我们可以说比起足球，网球的大小跟棒球更接近，但是无法在类别上做类似的区分——它们要么是同一类，要么不是。

类别型特征二值化后就变成了数值型特征，第3章中曾经用过。对于上述球的种类，可以创建三个二值特征："是网球""是棒球"和"是足球"。如果是网球的话，特征向量就是[1, 0, 0]，棒球[0, 1, 0]，足球[0, 0, 1]。这些特征都只有两个取值，很多算法都可以把它们作为连续型特征使用。二值化的好处是，便于直接进行数字上的比较（例如计算个体之间的距离）。

Adult数据集包含一些类别型特征，例如`Work-Class`（工作），其中它的有些值可以进行量级上的比较，比如不同的就业状况影响收入（例如有工作的人比没有工作的人收入高）。再比如，州政府职员的工资不太可能比私企职工工资高。

用数据框的`unique`函数就能得到所有的工作情况。

```
adult["Work-Class"].unique()
```

输出结果如下：

```
array([' State-gov', ' Self-emp-not-inc', ' Private', ' Federal-gov',
       ' Local-gov', ' ?', ' Self-emp-inc', ' Without-pay',
       ' Never-worked', nan], dtype=object)
```

数据集部分数据缺失，但不会影响这里的计算。

同理，数值型特征也可以通过离散化过程转换为类别型特征，第4章中曾用过。例如，高于1.7m的人，我们称其为高个子，低于1.7m的叫作矮个子。这样就得到了类别型特征（虽然是序数值类型）。在这个过程中确实丢失了一些信息。例如，有两个人，身高分别为1.69m和1.71m，这两个个子差不多的将被分到两个截然不同的类别里。而身高仅为1.2m的将会被认为与身高1.69m的"差不多高"！细节的丢失是离散化不好的一面，也是建模时需要考虑解决的问题。

对于Adult数据集，我们可以创建`LongHours`（时长）特征，用它来表示一个人每周工作时长是否多于40小时。这样就把连续值（`Hours-per-week`，每周工作时长）转换为类别型特征。

```
adult["LongHours"] = adult["Hours-per-week"] > 40
```

5.1.3　创建好的特征

建模过程中，需要对真实世界中的对象进行简化，这会导致信息的丢失，这也就是为什么没有一套能够用于任何数据集的通用的数据挖掘方法。数据挖掘的行家里手需要拥有数据来源领域的知识，没有的话，要积极去掌握。他们弄清楚问题是什么，了解有哪些可用数据后，在此基础上，才能创建解决问题所需的模型。

例如，身高这个特征描述的是一个人外表的某个方面，但是不能说明这个人学习成绩如何，所以预测学习成绩时，我们没必要去测量每个人的身高。

这正是数据挖掘为什么变得更像是一门艺术而不是科学的原因所在。抽取好的特征难度不小，该课题具有重要研究意义，因此它获得了研究人员的持续关注。选择好的分类算法也可以提升数据挖掘应用的效果，但最好还是通过选用好的特征来达到同样的目的。

不论是设计什么数据挖掘应用，首先都应该确定大致的方向，然后再设计方法实现目标。知道自己的目标之后，就能确定所需要的特征类型和算法，对最终结果也做到心中有数。

5.2 特征选择

通常特征数量很多，但我们只想选用其中一小部分。有如下几个原因。

- 降低复杂度：随着特征数量的增加，很多数据挖掘算法需要更多的时间和资源。减少特征数量，是提高算法运行速度，减少资源使用的好方法。
- 降低噪音：增加额外特征并不总会提升算法的表现。额外特征可能扰乱算法的正常工作，这些额外特征间的相关性和模式没有实际应用价值（这种情况在小数据集上很常见）。只选择合适的特征有助于减少出现没有实际意义的相关性的几率。
- 增加模型可读性：根据成千上万个特征创建的模型来解答一个问题，对计算机来说很容易，但模型对我们自己来说就晦涩无比。因此，使用更少的特征，创建我们自己可以理解的模型，就很有必要。

有些分类算法确实很强壮，能够处理噪音问题，特征再多也不在话下，但是选用干净的数据，选取更具描述性的特征，对算法效果提升很有帮助。

在开展数据挖掘工作前，有些基础性测试我们要做，比如确保特征值是不同的。如果特征值都相同，就跟没提供什么信息一样，挖掘就失去了意义。

scikit-learn中的VarianceThreshold转换器可用来删除特征值的方差达不到最低标准的特征。下面通过代码来讲下它的用法，首先用numpy创建一个简单的矩阵。

```
import numpy as np
X = np.arange(30).reshape((10, 3))
```

上述矩阵包含0到29，共30个数字，分为3列10行。可以把它看成一个有10个个体、3个特征的数据集。

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
```

```
        [12, 13, 14],
        [15, 16, 17],
        [18, 19, 20],
        [21, 22, 23],
        [24, 25, 26],
        [27, 28, 29]])
```

接着，把所有第二列的数值都改为1。

```
X[:,1] = 1
```

第一、三列特征值方差很大，而第二列方差为0。

```
array([[ 0, 1,  2],
       [ 3, 1,  5],
       [ 6, 1,  8],
       [ 9, 1, 11],
       [12, 1, 14],
       [15, 1, 17],
       [18, 1, 20],
       [21, 1, 23],
       [24, 1, 26],
       [27, 1, 29]])
```

这时再来创建`VarianceThreshold`转换器，用它处理数据集。

```
from sklearn.feature_selection import VarianceThreshold
vt = VarianceThreshold()
Xt = vt.fit_transform(X)
```

输出`Xt`后，我们发现第二列消失了。

```
array([[ 0,  2],
       [ 3,  5],
       [ 6,  8],
       [ 9, 11],
       [12, 14],
       [15, 17],
       [18, 20],
       [21, 23],
       [24, 26],
       [27, 29]])
```

输出每一列的方差。

```
print(vt.variances_)
```

下面输出结果表明第一、三列包含有价值信息，第二列方差为0，不包含具有区别意义的信息。

```
array([ 74.25, 0. , 74.25])
```

无论什么时候，拿到数据后，先做下类似简单、直接的分析，对数据集的特点做到心中有数。方差为0的特征不但对数据挖掘没有丝毫用处，相反还会拖慢算法的运行速度。

选择最佳特征

特征很多的情况下，怎么选出最佳的几个，可有点难度。它与解决数据挖掘问题自身相关，计算量很大。正如第4章讲到的，随着特征数量的增加，寻找子集的任务复杂度呈指数级增长。寻找最佳特征组合的时间复杂度同样是指数级增长的。

其中一个变通方法是不要找表现好的子集，而只是去找表现好的单个特征（单变量），依据是它们各自所能达到的精确度。分类任务通常是这么做的，我们一般只要测量变量和目标类别之间的某种相关性就行。

scikit-learn提供了几个用于选择单变量特征的转换器，其中SelectKBest返回*k*个最佳特征，SelectPercentile返回表现最佳的前r%个特征。这两个转换器都提供计算特征表现的一系列方法。

单个特征和某一类别之间相关性的计算方法有很多。最常用的有卡方检验（χ^2）。其他方法还有互信息和信息熵。

5

我们可以测试单个特征在Adult数据集上的表现。首先，选取下述特征，从pandas数据框中抽取一部分数据。

```
X = adult[["Age", "Education-Num", "Capital-gain", "Capital-loss",
  "Hours-per-week"]].values
```

接着，判断Earnings-Raw（税前收入）是否达到五万美元，创建目标类别列表。如果达到，类别为True，否则，类别为False。代码如下：

```
y = (adult["Earnings-Raw"] == ' >50K').values
```

再使用SelectKBest转换器类，用卡方函数打分，初始化转换器。

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
transformer = SelectKBest(score_func=chi2, k=3)
```

调用fit_transform方法，对相同的数据集进行预处理和转换。结果为分类效果较好的三个特征。代码如下：

```
Xt_chi2 = transformer.fit_transform(X, y)
```

生成的矩阵只包含三个特征。我们还可以得到每一列的相关性，这样就可以知道都使用了哪些特征。还是看下代码：

```
print(transformer.scores_)
```

输出结果如下：

```
[  8.60061182e+03   2.40142178e+03   8.21924671e+07   1.37214589e+06
   6.47640900e+03]
```

相关性最好的分别是第一、三、四列，分别对应着Age（年龄）、Capital-Gain（资本收益）和Capital-Loss（资本损失）三个特征。从单变量特征选取角度来说，这些就是最佳特征。

> 如果你想了解更多关于Adult数据集各特征的相关信息，请查看adult.names文件，里面有更多介绍及相关文献。

还可以用其他方法计算相关性，比如皮尔逊（Pearson）[①]相关系数。用于科学计算的SciPy库（scikit-learn在它基础上开发）实现了该方法。

> 如果你的计算机可以运行scikit-learn，那么就可以运行SciPy。运行下面这个例子不需要装额外的库。

首先，从SciPy导入pearsonr函数。

```
from scipy.stats import pearsonr
```

pearsonr函数的接口几乎与scikit-learn单变量转换器接口一致，该函数接收两个数组（当前例子中为x和y）作为参数，返回两个数组：每个特征的皮尔逊相关系数和p值（p-value）。前面用到的chi2函数使用的接口符合要求，因此我们直接把它传入到SelectKBest函数中。

SciPy的pearsonr函数参数为两个数组，但要注意的是第一个参数x为一维数组。我们来实现一个包装器函数，这样就能像前面那样处理多维数组。函数声明如下：

```
def multivariate_pearsonr(X, y):
```

创建scores和pvalues数组，遍历数据集的每一列。

```
    scores, pvalues = [], []
    for column in range(X.shape[1]):
```

只计算该列的皮尔逊相关系数和p值，并将其存储到相应数组中。

```
        cur_score, cur_p = pearsonr(X[:,column], y)
        scores.append(abs(cur_score))
        pvalues.append(cur_p)
```

① 卡尔·皮尔逊（Karl Pearson，1857—1936），原名Carl，英国数学家、生物统计学家。——译者注

> p值为−1到1之间的任意值。值为1，表示两个变量之间绝对正相关，值为−1，绝对负相关，即一个变量值越大，另一个变量值就越小，反之亦然。这样的特征确实能反应两个变量之间的关系，但是根据大小进行排序，这些值因为是负数而排在后面，可能会被舍弃不用。因此，我们对p值取绝对值后，再保存到scores数组中，而不是使用原始值。

函数最后返回包含皮尔逊相关系数和p值的元组。

```
    return (np.array(scores), np.array(pvalues))
```

现在，我们就可以像之前那样使用转换器类，根据皮尔逊相关系数对特征进行排序。

```
transformer = SelectKBest(score_func=multivariate_pearsonr, k=3)
Xt_pearson = transformer.fit_transform(X, y)
print(transformer.scores_)
```


返回的特征跟用卡方检验计算相关性得到的特征不一样！这回得到的是第一、二、五列：Age、Education和Hours-per-week。这表明哪些特征是最好的这个问题没有标准答案——取决于度量标准。

我们在分类器中看看哪个特征集合效果更好。请注意实验结果也只表明对于特定分类器和（或）特征子集效果更好——在数据挖掘领域，一种方法在任何情况下都比另一种方法好的情况几乎没有！实验代码如下：

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.cross_validation import cross_val_score
clf = DecisionTreeClassifier(random_state=14)
scores_chi2 = cross_val_score(clf, Xt_chi2, y, scoring='accuracy')
scores_pearson = cross_val_score(clf, Xt_pearson, y,
  scoing='accuracy')
```

chi2方法的平均正确率为0.83，而皮尔逊相关系数正确率为0.77。用卡方检验得到的特征组合效果更好！

别忘了本章数据挖掘任务目标是预测收入。使用特征选择技术，找到好的特征组合，只用三个特征，就能达到83%的正确率！

5.3 创建特征

有时，仅仅选择已有特征不够用。这时，我们就可以用不同的方法从已有特征中发掘新特征，比如前面用到的一位有效编码（one-hot encoding），我们可以用它把有A、B、C三个选项的类别型特征转换为三个新的特征："是A吗？""是B吗？"和"是C吗？"。

创建新特征看上去没必要，也没什么好处——毕竟，这些信息原本就在数据集里，我们只是要用而已。然而，特征之间相关性很强，或者特征冗余，会增加算法处理难度。

出于这个原因，创建特征就很有必要，很多方法也应运而生。

接下来，我们要加载一个新数据集，正好也该创建一个新的IPython Notebook笔记本文件了。从http://archive.ics.uci.edu/ml/datasets/Internet+Advertisements下载Advertisements（广告）数据集，保存到自己主目录下的Data文件夹中。

接着，用pandas加载数据集。我们还是先指定文件的路径。

```
import os
import numpy as np
import pandas as pd
data_folder = os.path.join(os.path.expanduser("~"), "Data")
data_filename = os.path.join(data_folder, "Ads", "ad.data")
```

数据集存在几个问题，加载过程需要我们做些处理。问题一，前几个特征是数值，但是pandas会把它们当成字符串。要修复这个问题，我们需要编写将字符串转换为数字的函数，该函数能够把只包含数字的字符串转换为数字，把其余的转化为“NaN”[①]（“Not a Number”，不是一个数字），表示参数值无法转换为数字，该值类似于其他编程语言中的none或null。

问题二，数据集中有些值缺失，缺失的值用“?”表示。幸运的是，问号不会被转换为浮点型数据，因此，我们也可以把它们转换为“NaN”。后续章节会讲到其他处理缺失值的方法。

转换函数声明如下：

```
def convert_number(x):
```

先试着把字符串转换为数字，如果失败就要处理异常，所以我们这里使用`try/except`结构，捕获`ValueError`异常（字符串无法转换为数字时，抛出异常）。

```
    try:
        return float(x)
    except ValueError:
```

在函数的最后，如果转换失败，返回`numpy`库中的“NaN”类型。

```
        return np.nan
```

我们来创建一个字典存储所有特征及其转换结果，我们想把所有的特征值转换为浮点型。

```
    converters = defaultdict(convert_number)
```

我们还想把最后一列（编号为#1558）的值转化为0或1，该列表示每条数据的类别。在Adult

① JavaScript语言中也有该类型，详见《JavaScript高级程序设计（第3版）》第29页。——译者注

数据集中，我们专门创建了一列表示类别。对于当前实验，导入数据时，顺便把类别这一列各个类别值由字符串转换为数值。

```
converters[1558] = lambda x: 1 if x.strip() == "ad." else 0
```

可以用read_csv加载数据集了，在参数中指定我们刚创建的转换函数。

```
ads = pd.read_csv(data_filename, header=None, converters=converters)
```

数据集很大，有1559列，2000多条数据。先来看下前五条数据，在笔记本的新格子中输入并运行ads[:5]。

	0	1	2	3	4	5	6	7	8	9	...	1549	1550	1551	1552	1553	1554	1555	1556	1557	1558
0	125	125	1.0000	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
1	57	468	8.2105	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
2	33	230	6.9696	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
3	60	468	7.8000	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
4	60	468	7.8000	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1

5

数据集所描述的是网上的图像，目标是确定图像是不是广告。

从数据集表头中无法获知每列数据的含义。你下载的另外两个文件ad.DOCUMENTATION和ad.names有更多信息。前三 个特征分别指图像的高度、宽度和宽高比。最后一列是数据的类别，1表示是广告，0表示不是广告。

其余特征取值为1的话，表示图像的URL、alt属性值或图像标题中包含某个特定的可以帮助判断是不是广告的词语，比如像sponsor（赞助商）。很多特征重合度很高，因为它们组合在一起恰好是其他特征。因此，数据集包含大量冗余信息。

将数据集加载到pandas之后，我们再来抽取用于分类算法的x矩阵和y数组，x矩阵为数据框除去最后一列的所有列，y数组包含数据框的最后一列，也就是列编号为#1558的列。代码如下。

```
X = ads.drop(1558, axis=1).values
y = ads[1558]
```

主成分分析

有些数据集，特征之间联系紧密。例如，只有1个档的微型单座赛车，速度和油耗两者关系就极为密切。特征间的相关性信息在某些场合会很有用，但是数据挖掘算法通常不需要这些冗余信息。

Advertisements数据集中有些特征相关性特别大，因为很多关键词在图像的alt属性值及图像的标题中都会使用。

主成分分析算法（Principal Component Analysis，PCA）的目的是找到能用较少信息描述数据集的特征组合。它意在发现彼此之间没有相关性、能够描述数据集的特征，确切说这些特征的

方差跟整体方差没有多大差距，这样的特征也被称为主成分。这也就意味着，借助这种方法，就能通过更少的特征捕获到数据集的大部分信息。

PCA跟其他转换器用法类似。它只有主成分数量这一个参数。它默认会返回数据集中的所有特征。然而，PCA会对返回结果根据方差大小进行排序，返回的第一个特征方差最大，第二个特征方差稍小，以此类推。因此，前几个特征往往就能够解释数据集的大部分信息。请见代码：

```
from sklearn.decomposition import PCA
pca = PCA(n_components=5)
Xd = pca.fit_transform(X)
```

返回的结果Xd矩阵只有五个特征，但是不容小觑，我们看一下每个特征的方差。

```
np.set_printoptions(precision=3, suppress=True)
pca.explained_variance_ratio_
```

结果array([0.854, 0.145, 0.001, 0. , 0.])表明第一个特征的方差对数据集总体方差的贡献率为85.4%。第二个为14.5%。第四个特征方差贡献率不可能高于1‰，后面1553个特征则更少。

用PCA算法处理数据一个不好的地方在于，得到的主成分往往是其他几个特征的复杂组合，例如，上述第一个特征就是通过为原始数据集的1558个特征（虽然很多特征值为0）分别乘以不同权重得到的，前三个特征的权重依次为–0.092、–0.995和–0.024。经过某种组合得到的特征，如果没有丰富的研究经验，理解起来很困难。

用PCA算法得到的数据创建模型，不仅能够近似地表示原始数据集，还能提升分类任务的正确率。

```
clf = DecisionTreeClassifier(random_state=14)
scores_reduced = cross_val_score(clf, Xd, y, scoring='accuracy')
```

正确率为0.9356，比起只用所有的原始特征效果（稍）好。PCA算法不是说都能带来效果上的提升，但多数情况是有好处的。

> 我们这里用PCA算法来减少数据集中的冗余特征。一般来说，你不能用它来解决数据挖掘实验的过拟合问题。原因是PCA没有将类别考虑进去。更佳解决方案是正则化，简介及代码请见 http://blog.datadive.net/selecting-good-features-part-ii-linear-models-and-regularization/。

PCA算法的另一个优点是，你可以把抽象难懂的数据集绘制成图形。例如，把PCA返回的前两个特征做成图形。

首先，告诉IPython在当前笔记本作图，导入pyplot。

```
%matplotlib inline
from matplotlib import pyplot as plt
```

接着，获取数据集中类别的所有取值（只有两个：是广告和不是广告）。

```
classes = set(y)
```

指定在图形中用什么颜色表示这两个类别。

```
colors = ['red', 'green']
```

用zip函数将这两个列表组合起来，同时遍历。

```
for cur_class, color in zip(classes, colors):
```

为属于当前类别的所有个体创建遮罩层。

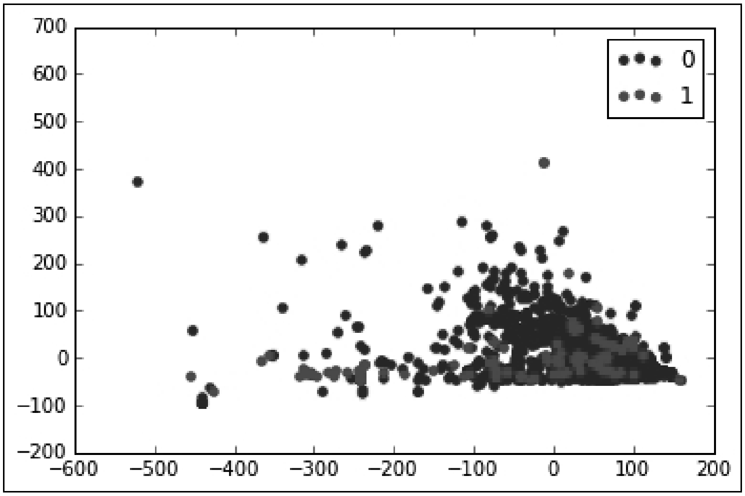
```
    mask = (y == cur_class).values
```

使用pyplot的scatter函数显示它们的位置。图中的x和y的值为前两个特征。


```
    plt.scatter(Xd[mask,0], Xd[mask,1], marker='o', color=color,
      label=int(cur_class))
```

最后，在循环的外面，创建图示，显示图像，从中就能看到分属两个类别的个体。

```
plt.legend()
plt.show()
```



5.4 创建自己的转换器

随着数据集复杂程度的提高和类型的变化，你可能发现，现成的特征抽取转换器不能满足需

求了。比如，第7章从图这种数据结构里抽取特征，就需要自己编写转换器。

转换器像极了转换函数。它接收一种形式的数据，输出另外一种形式。转换器可以用训练集训练，训练得到的参数可以用来转换测试数据集。

转换器的API（应用编程接口）很简单。它接收一种特定格式的数据，输出一种格式的数据（可能与输入数据的格式相同，也可能不同）。别的就没有什么特殊要求了。

5.4.1　转换器 API

转换器有两个关键函数。

- `fit()`：接收训练数据，设置内部参数。
- `transform()`：转换过程。接收训练数据集或相同格式的新数据集。

`fit()`和`transform()`函数的输入应该为同一种数据类型，但是`transform()`可以返回不同的数据类型。

我们现在就通过实现一个简单的转换器，来说明下API的使用。转换器接收`numpy`数组作为输入，根据均值将其离散化。任何高于均值的特征值（训练集中）替换为1，小于或等于均值的替换为0。

我们曾经用pandas对Adult数据集做过类似的转换：用`Hours-per-week`特征创建了`LongHours`特征，后者表示每周工作时长是否在40小时以上。即将编写的转换器有两个不同点。第一，接口与`scikit-learn`接口一致，便于在流水线中使用。第二，使用均值而不是固定的值（在`LongHours`例子中使用40作为将特征二值化的阈值）。

5.4.2　实现细节

打开分析Adult数据集时用到的笔记本文件。点击Cell 菜单，选择Run All，运行所有单元格。

首先，导入用来设置API 的`TransformerMixin`类。Python语言（相对于Java之类的语言）没有严格的接口规范，使用类似`mixin`这样的类，`scikit-learn`就会把我们封装的类当作是转换器。我们还需要导入用来判断输入类型是否合法的函数，稍后就会用到。

导入所需模块。

```
from sklearn.base import TransformerMixin
from sklearn.utils import as_float_array
```

接着，创建继承自`mixin`的类。

```
class MeanDiscrete(TransformerMixin):
```

我们需要定义接口规范与API一致的fit和transform函数。在fit函数中，计算出数据集的均值，用内部变量保存该值。函数声明如下：

```
    def fit(self, X):
```

fit函数中，要确保X数据集可以处理，我们用到了as_float_array函数（尝试对X进行转换，例如X是浮点数列表时就可以对其进行转换）。

```
        X = as_float_array(X)
```

计算数组的均值，用内部变量保存该值。如果X是多变量数组，数组self.mean保存的是每个特征的均值。

```
        self.mean = X.mean(axis=0)
```

fit函数需要返回它本身，确保在转换器中能够进行链式调用（例如调用ransformer.fit(X).transform(X)）。函数返回语句如下：

```
        return self
```


接着，新建transform函数，参数类型与fit函数相同，检查下输入是否合法。

```
    def transform(self, X):
        X = as_float_array(X)
```

输入必须是numpy数组（或等价的数据结构），除此之外，还需要检查输入的数据列数是否一致。X中的特征数应该与训练集中的特征数一致。

```
        assert X.shape[1] == self.mean.shape[0]
```

函数最后返回X中大于均值的数据。

```
        return X > self.mean
```

初始化一个实例，用它来对X数组进行转换。

```
mean_discrete = MeanDiscrete()
X_mean = mean_discrete.fit_transform(X)
```

5.4.3 单元测试

创建完函数和类后，最好是做下单元测试。单元测试，顾名思义，就是对代码中完成一个功能的代码单元进行测试。在这里，测试下转换器的功能是否符合预期。

充分的测试应当在独立于现有代码的前提下进行。为保证测试的有效性，最好使用另一种编程语言或方法来进行相同的计算。我们这里使用Excel创建数据集，再计算每一列的均值，接下

来跟程序计算结果相比较。

单元测试应该短小、运行速度快。因此，我们选用少量数据进行测试。本例用于测试的数据集保存在前面创建的Xt变量中，在测试中会重新创建这个数据集。该数据集两个特征的均值分别为13.5和15.5。

为了创建单元测试，我们从numpy的testing模块中导入assert_array_equal函数，它用来检测两个数组是否相等。

```
from numpy.testing import assert_array_equal
```

接下来，创建测试函数，注意文件名以“test_”开头，这种命名方法便于工具自动查找并运行测试。代码如下：

```
def test_meandiscrete():
    X_test = np.array([[ 0,  2],
                        [ 3,  5],
                        [ 6,  8],
                        [ 9, 11],
                        [12, 14],
                        [15, 17],
                        [18, 20],
                        [21, 23],
                        [24, 26],
                        [27, 29]])
```

创建转换函数实例，用测试数据进行训练。

```
    mean_discrete = MeanDiscrete()
    mean_discrete.fit(X_test)
```

通过与在Excel中得到的结果比较，检查内部均值参数是否正确设置。

```
    assert_array_equal(mean_discrete.mean, np.array([13.5, 15.5]))
```

运行转换函数，创建被转换过后的数据集。准备好测试数据（在Excel中创建的）。

```
    X_transformed = mean_discrete.transform(X_test)
    X_expected = np.array([[ 0,  0],
                            [ 0, 0],
                            [ 0, 0],
                            [ 0, 0],
                            [ 0, 0],
                            [ 1, 1],
                            [ 1, 1],
                            [ 1, 1],
                            [ 1, 1],
                            [ 1, 1]])
```

最后，测试转换器返回结果是否与期望结果一致。

```
assert_array_equal(X_transformed, X_expected)
```

调用函数，开始测试！

```
test_meandiscrete()
```

如果没弄错，上述代码应该没有问题！可以改动测试数据，用几个错误值，看看是否测试失败。要让测试结果通过，记得再改回正确的值。

如果要进行多个测试，最好使用nose测试框架来运行测试。

5.4.4 组装起来

现在，我们有了一个经过测试的转换器，拿出来用用吧。用上我们之前学到的知识，创建一个流水线，第一步使用MeanDiscrete转换器，第二步使用决策树分类器，然后进行交叉检验，输出结果。来看下代码：


```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([('mean_discrete', MeanDiscrete()),
  ('classifier', DecisionTreeClassifier(random_state=14))])
  scores_mean_discrete = cross_val_score(pipeline, X, y,
    scoring='accuracy')
  print("Mean Discrete performance:
{0:.3f}".format(scores_mean_discrete.mean()))
```

正确率为0.803，结果没有之前高，对于简单的二值特征而言还算不错。

5.5 小结

本章重点为特征和转换器，以及如何将其用到数据挖掘流水线中。我们探讨了什么是好特征，以及如何用算法从数据集中选取最佳特征。然而，创建好的特征比起科学更像是一门艺术，通常需要领域相关的知识和经验。

我们创建了自己的转换器，接口与scikit-learn常用工具接口一致。后续章节会创建更多的转换器，可以使用现有函数对其进行充分测试。

下一章将介绍文本的特征抽取方法。对于文本而言，转换器和特征类型都不少，但它们各有利弊。

第 6 章

使用朴素贝叶斯进行社会媒体挖掘

不论是书、历史文档、社会媒体、电子邮件还是其他以文字为主的通信方式，都包含大量信息。从文本数据集抽取特征，用于分类不是件容易事。然而，人们还是总结出了文本挖掘的通用方法。

本章介绍如何用强大却出奇简单的朴素贝叶斯算法消除社会媒体用语的歧义。朴素贝叶斯算法在计算用于分类的概率时，为简化计算，假定各特征之间是相互独立的，因此名字中含有朴素二字。它稍经扩展就能用于对其他类型数据集进行分类，且不依赖于数值特征。本章创建的模型在多种数据集上表现还不错，可作为很多文本挖掘研究的基准。

本章主要涉及如下内容。

- 用社交网络的API下载数据
- 用于处理文本的转换器
- 朴素贝叶斯分类器
- 用JSON保存和加载数据集
- 用NLTK库从文本中抽取特征
- 用F值评估分类效果

6.1 消歧

文本通常被称为无结构格式，虽然它包含很多信息，但是却没有标题、特定格式，句法松散，以及其他问题，导致难以从中提取有用信息。数据间联系紧密，行文中经常相互提及，交叉引用现象也很常见——从这种格式中提取信息难度很大！

我们通过比较书中的信息和大型数据库中的信息，来看下两者都有哪些方面的不同。书中有角色、主题、场所等大量信息。然而，只有去读书，并且光读还不行，更重要的是理解后才能获

得书中的信息。而位于服务器数据库的数据，每一列都有名字，都有指定的数据类型。所有的信息都在数据库中，解释起来很容易。描述数据类型、含义的信息叫作*元数据*，文本中缺乏这类数据。书中的目录和索引虽含有部分元数据，但是比起数据库对于数据精确的定义和描述，这点元数据实在微不足道。

文本挖掘的一个难点来自于歧义，消除歧义常被简称为*消歧*。当人们使用bank[①]这个词时，他透露的是金融相关的信息还是环境相关的（比如河岸）？在很多情况下，对我们自己来说，消除歧义比较容易（有时也不简单），但是对计算机来说难度要大得多。

本章将探讨如何区别Twitter消息中Python的意思。Twitter网站上的一条消息叫作tweet，它最多不能超过140个字符。这也就表明上下文信息较少。此外，没有多少元数据，虽然“#”号经常用来表示消息的主题。

当人们提及Python时，他们谈论的可能是：

- 编程语言Python
- 经典喜剧剧团Monty Python[②]
- 蟒蛇
- 鞋子品牌Python

可能还有很多其他东西也叫Python。我们实验的目的是根据消息的内容，判断消息中的Python是不是指编程语言。

6.1.1 从社交网站下载数据

接下来，从Twitter网站下载一些语料，从中剔除垃圾信息后，用于分类任务。Twitter提供了从他们服务器采集信息的强大API，小规模使用免费，但如果你打算将Twitter数据用于商业用途，请了解下相关规定。

首先，你需要一个Twitter账号（免费）。如果没有的话，请访问http://twitter.com进行注册。

每分钟的请求数不能超过Twitter所规定的上限。写作本书时，上限为每小时180次。这个规定不好遵照执行，因此，强烈建议使用现成的库与Twitter的API进行通信。

访问Twitter数据时需要提供密钥。打开http://twitter.com，登录Twitter。

登录后，访问https://apps.twitter.com/，点击Create New App（创建新应用）。

① 在英语中，bank既可以指银行，也可以指河岸。——译者注

② 编程语言Python的名字正是来自于这个英国喜剧剧团的名字，Guido van Rossum很喜欢该剧团的演出，于是把他创立的编程语言命名为Python。——译者注

指定新应用的名称，填好描述及要在哪个网站中使用。如果不打算在网站中使用，请在Website文本框中随意输入些内容，确保提交时表单能通过验证。Callback URL文本框空着不填——我们用不到。在下面的开发者协议处选中Yes, I agree前面的复选框（如果你确实同意），点击Create your Twitter application。

创建新应用后，先别急着关掉当前页面——后面会用到页面上的access keys（访问密钥）。接下来，需要找一个与Twitter通信的Python第三方库。选择有很多，我喜欢用Twitter官方提供的`twitter`库。

> 如果你习惯用`pip`安装第三方包，可以使用`pip3 install twitter`安装`twitter`库。如果你用的是其他系统，安装方法请参考文档https://github.com/sixohsix/twitter。

创建新的IPython笔记本文件来编写下载Twitter消息的代码。本章将创建几个不同的笔记本文件，用于不同的处理任务，所以最好是新建一个文件夹，把它们放到一起。第一个笔记本文件ch6_get_twitter专门用来下载新的Twitter语料。

首先，导入`twitter`库，设置授权令牌。刚才没让你关闭的那一页的Keys and Access Tokens（密钥和访问令牌）选项卡下，有consumer key（用户密钥）和consumer secret（请求令牌）。点击在同一页的Create my access token（创建我的访问令牌）按钮，获取访问令牌。用你得到的密钥和令牌替换下面代码中的占位符。

```
import twitter
consumer_key = "<Your Consumer Key Here>"
consumer_secret = "<Your Consumer Secret Here>"
access_token = "<Your Access Token Here>"
access_token_secret = "<Your Access Token Secret Here>"
authorization = twitter.OAuth(access_token, access_token_secret,
consumer_key, consumer_secret)
```

我们将用`twitter`库提供的搜索函数（`search`）查找包含单词“Python”的消息。创建阅读器对象，提供授权信息连接到`Twitter`，然后使用阅读器对象进行搜索。在笔记本文件中，指定消息的存储位置。

```
import os
output_filename = os.path.join(os.path.expanduser("~"),
  "Data", "twitter", "python_tweets.json")
```

保存数据时要用到`json`库。

```
import json
```

接着，创建用来从Twitter网站读取数据的对象，指定授权信息，需要用到前面创建的授权对象。

```
t = twitter.Twitter(auth=authorization)
```

打开输出文件，等待写入数据，写入模式指定为“a”，这样每次在文件末尾追加新数据。使用建立好的连接，搜索单词“Python”，对于search方法返回的数据，我们只需要“statuses”部分。下面代码获取到Twitter消息，使用json库的dump函数将其转换为字符串形式后，写入到输出文件中。写完每条消息后，再写一行空行，便于把每条消息区分开来。

```
with open(output_filename, 'a') as output_file:
    search_results = t.search.tweets(q="python", count=100)['statuses']
    for tweet in search_results:
        if 'text' in tweet:
            output_file.write(json.dumps(tweet))
            output_file.write("\n\n")
```

上述循环中，还检测了消息是否包括“text”键。并不是Twitter返回的所有对象都是消息（有些可能是用来删除消息或其他内容的动作）。消息对象与其他对象的关键不同在于消息对象中含有键“text”，这也正是我们用if语句进行检测的。

上述代码运行几分钟后，输出文件中就能有100条消息。

> 你也可以让它多运行几分钟，抓取更多的消息，需要注意的是，如果Twitter用户没有发布新消息的话，短时间内多次抓取，得到的消息很可能会有重复的。

6.1.2 加载数据集并对其分类

完成消息采集后，我们拿到了原始数据集，对它里面的消息逐条标注后，才能用于分类任务。在笔记本文件中创建一个表单，方便标注。

消息存储格式近似于JSON。JSON格式不会对数据强加过多用于表示结构的信息，可以直接用JavaScript（JSON名字也就是这么来的，JavaScript Object Notation，JavaScript对象表示法）语言读取。JSON定义了诸如数字、字符串、数组、字典等基本对象，适合存储包含非数值类型的数据。如果数据集全都是数值类型，为了节省空间和时间，最好使用类似于numpy矩阵这样的格式来存储。

我们的数据集格式和真正的JSON对象的关键区别在于，每两条消息之间有一行空行。这样做的目的是，防止新追加的消息和前面的消息混在一起（在真正的JSON格式中，追加数据没这么简单）。我们的数据集中，每两条用JSON字符串表示的消息之间有一行空行。

我们可以使用json库解析数据集，但是要先根据空行把读进来的文件拆分（split方法）成一个列表，得到真正的消息对象。

新建一个笔记本文件（我把它命名为ch6_label_twitter），指定数据集的名称。该名称即为上

节指定的输出文件的名字。我们还指定用于存放每条消息所属类别的文件名。代码如下：

```
import os
input_filename = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "python_classes.json")
```

上面已经提到过，我们要使用json库，先来导入它。

```
import json
```

创建列表，用于存储从文件中读进来的每条消息。

```
tweets = []
```

遍历文件中的每一行数据。我们对不包含消息的空行（它们用于分隔消息）不感兴趣，因此，检测当前行（去除任意空白字符后的）长度是否为0。如果为0，忽略当前行，继续判断下一行。如果不为0，使用json.loads（将JSON字符串转换为Python对象）方法加载消息，将它添加到tweets列表中。代码如下：

```
with open(input_filename) as inf:
    for line in inf:
        if len(line.strip()) == 0:
            continue
        tweets.append(json.loads(line))
```

我们现在想知道一条消息是否和我们相关（在这里，相关表示指的是编程语言Python）。接下来，在笔记本文件中嵌入HTML代码，实现JavaScript和Python之间的通信，用网页形式展示消息，便于标注。

我们要实现以下功能，向用户（你）展示一条消息，要求用户输入类别：相关还是不相关。程序保存输入结果，继续展示下一条待标注的消息。

首先，创建用于存储类别（标注结果）的列表。不管消息是不是与编程语言Python相关，我们都保存它的类别，分类器从这两类数据中学习如何预测一条消息的类别。

我们还要检测是不是有部分消息已经标注过类别了，有的话就加载这些类别。如果你标注到一半，临时有事要关闭笔记本文件，有了该功能，再打开时，代码就会加载已有类别。一般来说，对于类似的任务，考虑如何保存中间结果很有必要。这样即使计算机中途死机，努力一个小时得来的工作成果也不至于全部丢失！代码如下：

```
labels = []
if os.path.exists(labels_filename):
    with open(labels_filename) as inf:
        labels = json.load(inf)
```

接下来，创建一个简单的函数，用来返回下一条需要标注的消息。我们找到并返回第一条没

有标注类别的消息即可。代码如下。

```
def get_next_tweet():
    return tweet_sample[len(labels)]['text']
```

> 我们实验的下一个步骤是收集用户（你！）对每条消息中的“Python”是否指的是编程语言的看法。在笔记本文件中，仅使用Python，无法通过直接与用户进行交互的方式来收集他们的反馈。因此，我们只好使用一点JavaScript和HTML代码来获取用户输入。

接下来，在笔记本中创建JavaScript程序来收集输入。可以借助魔术方法（magic function）在笔记本中直接嵌入HTML和JavaScript代码等。在笔记本的新格子中输入如下代码。

```
%%javascript
```

这样就表示下面为JavaScript代码，因此，大括号就要登场了。别担心，我们很快就会再回到Python的。请注意下面JavaScript代码必须与魔术方法`%%javascript`在同一格子里。

从下面定义的第一个JavaScript函数，就能看到在笔记本中，实现JavaScript和Python之间的通信是多么容易。这个函数的功能是向`labels`列表（Python代码中）添加一条消息所属的类别。具体做法是先加载IPython内核（kernel）对象，再用它来执行Python命令。代码如下：

```
function set_label(label){
    var kernel = IPython.notebook.kernel;
    kernel.execute("labels.append(" + label + ")");
    load_next_tweet();
}
```

函数最后调用`load_next_tweet`函数，加载下一条未标注的消息。`load_next_tweet`这个JavaScript函数内部执行Python代码的原理跟上面所讲的相同：用加载的IPython内核来执行Python命令（调用前面定义的`get_next_tweet`函数）。

然而，获取并展示消息有点困难，需要用到回调函数，返回数据时调用该函数。回调函数的定义方法超出了本书的范围。如果你对更高级的JavaScript/Python交互方法感兴趣，请参考IPython文档。

代码如下：

```
function load_next_tweet(){
   var code_input = "get_next_tweet()";
   var kernel = IPython.notebook.kernel;
   var callbacks = { 'iopub' : {'output' : handle_output}};
   kernel.execute(code_input, callbacks, {silent:false});
}
```

回调函数叫作handle_output，下面就来创建它。当kernel.execute()调用的Python函数返回结果后，就会调用回调函数。如上所述，回调函数的详细内容不在本书讲解范围之列，我们这里用它来处理返回的纯文本格式数据，把这些数据置于表单的#tweet_text div中展示，我们随后会编写相关HTML代码。回调函数代码如下：

```
function handle_output(out){
   var res = out.content.data["text/plain"];
   $("div#tweet_text").html(res);
}
```

HTML代码中，最外层是id为tweetbox的div，它里面包着id为tweet_text的div元素用于显示下一条待标注的消息。我们还创建文本框，捕获输入的按键（否则，笔记本程序将会捕获它，JavaScript也就无法获得用户的输入）。这样我们就可以用键盘来设置消息的类别为1或0，比起用鼠标点击按钮进行选择要快——假如我们至少需要标注100条消息。

运行JavaScript代码所在的格子，就会在页面中嵌入JavaScript代码，虽然在结果区域看不到任何变化。

接着，我们来使用另一个魔术方法%%html。毫无疑问，它是用来直接在笔记本中嵌入HTML代码。在新格子中输入如下代码。

```
%%html
```

接下来，在这个格子中输入HTML代码和几行JavaScript代码。首先，定义一个div元素，用来显示当前要标注的消息。我还添加了几行标注说明。接着，创建id为tweet_text的div元素，用来显示下一条待标注的消息。如前所述，我们还需要创建文本框用来捕获按键。代码如下：

```
<div name="tweetbox">
    Instructions: Click in textbox. Enter a 1 if the tweet is
relevant, enter 0 otherwise.<br>
Tweet: <div id="tweet_text" value="text"></div><br>
<input type=text id="capture"></input><br>
</div>
```

先不要运行这个格子！

创建完表单后，我们来编写捕获键盘按键的JavaScript代码，这段脚本须写到上面HTML代码的后面，因为#tweet_text元素在HTML代码运行前在页面上还不存在。这里要用到jQuery库（IPython笔记本文件使用了该库，无需再次引入）的一个函数，当在#capture文本框元素上发生按键事件时，调用指定的函数。然而，请注意，这个格子使用的魔术方法是%%html，在这里面写JavaScript代码，需要将其放到<script>标签中。

我们只关注按键0或1，因为消息只可能属于这两个类别。通过检测存储在e.which中的ASCII码值，就能确定到底是哪个键被按下了。如果用户按下0或1，我们把类别添加到labels列表中，

然后清空文本框的值。代码如下：

```
<script>
$("input#capture").keypress(function(e) {
if(e.which == 48) {
    set_label(0);
    $("input#capture").val("");
}else if (e.which == 49){
    set_label(1);
    $("input#capture").val("");
  }
});
```

忽略其他按键。

下面是本章最后几行JavaScript代码（我向你保证），我们调用`load_next_tweet()`函数。设置第一条待标注的消息，闭合`script`标签。代码如下：

```
load_next_tweet();
</script>
```

在笔记本中，运行该格子，页面上会出现HTML文本框，旁边就是第一条消息。点击文本框，如果该消息与我们相关（*消息中的Python指的是编程语言*），输入1；反之，输入0。完成后，加载下一条消息。输入类别，接着加载下一条。重复以上过程，直到标注完所有数据。

完成标注后，把所有的类别信息输出到前面定义好的类别文件中。

```
with open(labels_filename, 'w') as outf:
    json.dump(labels, outf)
```

即使没有完成标注，也可以运行上述代码，保存标注过的类别。再次运行笔记本将会加载你没有标注的消息，这样你就能继续标注。

标注消息要花点时间！消息很多的话，更是如此。如果你为了赶时间，可以下载我标注好的数据集。

6.1.3 Twitter 数据集重建

数据挖掘过程会用到很多变量，它们不仅出现在挖掘算法里，数据采集等过程中也少不了它们的身影。重现实验结果很重要，因为它有助于验证或改善实验效果。[①]

① 这一段话，照直翻译过来，总觉得缺了点什么，推测作者想要表达的意思是控制每次实验中变量的取值，确保实验结果的可比较性。——译者注

> 算法X在一个数据集上取得80%的正确率，算法Y在另一个数据集上取得90%的正确率，不能说明算法Y优于X。只有在相同的测试集上，且在相同的条件下进行测试，才能比较算法的优劣。

你用上面的代码采集到的数据集与我的不同。主要原因在于，采集时间不同，Twitter返回的搜索结果也就不同。此外，标注结果也可能会有所差异。虽然有些消息显然是与编程语言Python有关，但也不乏模棱两可的情况，比如用我不懂的外语发布的消息，我给出的标注结果就可能有误。对于这种情况，Twitter提供了设置语言种类的接口，但即使使用该接口，返回的消息也不一定全都是你预先指定的语言。

出于这么多原因，对于从社交网站采集到的数据集重复进行相同的实验困难重重，Twitter也不例外。此外，Twitter明确禁止直接分享数据集。

解决方法之一是只分享消息编号，它是可以自由传播的。本节首先创建可以自由分享的消息编号数据集。然后，再来看下如何根据编号下载消息，重建先前的数据集。

首先，我们得把现有消息的编号及其类别保存下来。创建一个新的笔记本文件，指定接下来要用到的几个文件名。代码跟之前类似，只不过多了一个用来保存消息编号及其类别的文件。代码如下：

```
import os
input_filename = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "python_classes.json")
replicable_dataset = os.path.join(os.path.expanduser("~"),
  "Data", "twitter", "replicable_dataset.json")
```

加载消息和类别，就跟我们在上一个笔记本文件中做的那样。

```
import json
tweets = []
with open(input_filename) as inf:
    for line in inf:
        if len(line.strip()) == 0:
            continue
        tweets.append(json.loads(line))
if os.path.exists(labels_filename):
    with open(classes_filename) as inf:
        labels = json.load(inf)
```

同时遍历所有的消息及消息所属的类别，创建新数据集，将其保存到列表中。

```
dataset = [(tweet['id'], label) for tweet, label in zip(tweets,
  labels)]
```

最后，把结果保存到文件中。

```
with open(replicable_dataset, 'w') as outf:
    json.dump(dataset, outf)
```

有了消息的编号和类别，我们就可以重建数据集。如果你想重建我在本章使用的数据集，所需代码请见本书配套代码包。

加载之前的数据集不难，但是要花些时间。新建一个笔记本文件，像之前那样指定好用于存储数据集、消息类别和消息编号的文件。我调整了下文件名，防止你覆盖掉之前采集的数据集，文件名你可以随意起，不必跟我的一样。代码如下：

```
import os
tweet_filename = os.path.join(os.path.expanduser("~"), "Data",
  "twitter", "replicable_python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data",
  "twitter", "replicable_python_classes.json")
replicable_dataset = os.path.join(os.path.expanduser("~"),
  "Data", "twitter", "replicable_dataset.json")
```

使用JSON从文件中加载消息编号及类别数据。

```
import json
with open(replicable_dataset) as inf:
    tweet_ids = json.load(inf)
```

保存所有消息的类别很容易。遍历数据集，抽取编号，用两行代码就能搞定（打开文件和保存消息）。然而，我们无法确定之后能再次获取到所有消息（例如，上次采集后，有些消息被设置为隐私），因此类别和消息可能就无法对应。

为了举例说明，我在采集数据后的第一天尝试重建数据集，却发现有两条消息已经从线上消失（可能被用户删除或设置为隐私）。因此，只输出我们实际能用到的类别就显得尤为重要。具体做法是，首先，创建`actual_labels`列表存储我们能够再次从Twitter网站获取到的消息的类别。然后，创建字典，为消息的编号和类别建立起映射关系。

代码如下：

```
actual_labels = []
label_mapping = dict(tweet_ids)
```

接下来，用`twitter`库根据消息编号采集消息。这可能要花点时间。导入前面用过的`twitter`库，创建授权令牌，用它来初始化`twitter`对象。

```
import twitter
consumer_key = "<Your Consumer Key Here>"
consumer_secret = "<Your Consumer Secret Here>"
access_token = "<Your Access Token Here>"
access_token_secret = "<Your Access Token Secret Here>"
```

6

```
authorization = twitter.OAuth(access_token, access_token_secret,
  consumer_key, consumer_secret)
t = twitter.Twitter(auth=authorization)
```

遍历并抽取所有的消息编号。

```
all_ids = [tweet_id for tweet_id, label in tweet_ids]
```

然后，打开输出文件，保存消息。

```
with open(tweets_filename, 'a') as output_file:
```

Twitter API允许我们一次只能获取100条消息。因此，每次遍历100条消息。

```
    for start_index in range(0, len(tweet_ids), 100):
```

把这一批次的100个编号用逗号连接起来，便于下面使用Twitter的API根据编号查找消息。

```
        id_string = ",".join(str(i) for i in
          all_ids[start_index:start_index+100])
```

接着，调用Twitter定义的`statuses/lookup`方法，传入一批消息编号（已转换为字符串），以采集这些消息。

```
        search_results = t.statuses.lookup(_id=id_string)
```

我们把返回结果中的每一条消息，按照之前采集数据集的做法，将它们依次保存到文件中。

```
        for tweet in search_results:
            if 'text' in tweet:
                output_file.write(json.dumps(tweet))
                output_file.write("\n\n")
```

最后一步（仍然属于`if`模块），还需要保存当前遍历到的消息的类别。获取消息类别要用到之前创建的`label_mapping`字典，根据消息编号查找即可。代码如下：

```
                actual_labels.append(label_mapping[tweet['id']])
```

运行上述代码，采集所有消息。如果你的数据集很大，花费时间相应较多——Twitter会限制请求的频次。最后一步，保存`actual_labels`到类别文件[①]里。

```
with open(labels_filename, 'w') as outf:
    json.dump(actual_labels, outf)
```

6.2　文本转换器

数据集创建好后，怎样在它上面施展数据挖掘的威力呢？

① 再次提醒下它里面存放的是能够再次抓取到的消息的类别。——译者注

文本数据集包括图书、文章、网站、手稿、代码以及其他形式的文本。我们目前所见过的所有算法不是用来处理数值型就是类别型特征，怎样才能把文本转换成算法可以处理的形式？

多种测量方法能够帮上忙。比如，平均词长和平均句长可用来预测文本的可读性。除此之外，还有很多其他类型的特征，比如我们接下来要用到的单词是否出现（word occurrence）。

6.2.1　词袋

一种最简单却非常高效的模型就是只统计数据集中每个单词的出现次数。我们来创建一个矩阵，每一行表示数据集中的一篇文档，每一列代表一个词。矩阵中的每一项为某个词在文档中的出现次数。

下面这段文字节选自托尔金（J. R. R. Tolkien）的《指环王》。

Three Rings for the Elven-kings under the sky,

Seven for the Dwarf-lords in halls of stone,

Nine for Mortal Men, doomed to die,

One for the Dark Lord on his dark throne

In the Land of Mordor where the Shadows lie.

One Ring to rule them all, One Ring to find them,

One Ring to bring them all and in the darkness bind them.

In the Land of Mordor where the Shadows lie.

—*J.R.R. Tolkien's epigraph to The Lord of The Rings*

单词the在引文中出现了9次，单词in、for、to和one各出现了4次。单词ring和of各出现3次。

从中选取几组数据，创建一个简单的数据集。

单词	the	one	ring	to
频次	9	4	3	4

可以用counter方法统计列表中各字符串出现次数。统计单词时，通常将所有字母转换为小写，因此定义字符串时顺便把它转换为小写形式。代码如下：

```
s = """Three Rings for the Elven-kings under the sky,
Seven for the Dwarf-lords in halls of stone,
Nine for Mortal Men, doomed to die,
```

6

```
One for the Dark Lord on his dark throne
In the Land of Mordor where the Shadows lie.
One Ring to rule them all, One Ring to find them,
One Ring to bring them all and in the darkness bind them.
In the Land of Mordor where the Shadows lie. """.lower()
words = s.split()
from collections import Counter
c = Counter(words)
```

`c.most_common(5)`输出出现次数最多的前5个词，竟然有4个词并列第二，多输出几个就能看出词频的差异了。

词袋模型主要分为以下三种：第一种像上面这样使用词语实际出现次数作为词频。缺点是当文档长度差异明显时，词频差距会非常大。第二种是使用归一化后的词频，每篇文档中所有词语的词频之和为1。这种做法优势明显，它规避了文档长度对词频的影响。第三种，直接使用二值特征来表示——单词在文档中出现值为1，不出现值为0。本章使用第三种。

另外一种（更）通用的规范化方法叫作*词频–逆文档频率法*（term frequency-inverse document frequency，简写为tf-idf），该加权方法用词频来代替词的出现次数，然后再用词频除以包含该词的文档的数量。第10章将会用到词频–逆文档频率法。

Python有很多用于处理文本的库。我们将使用主流的NLTK库（Natural Language ToolKit，自然语言处理工具集）抽取特征。`scikit-learn`提供进行类似处理的`CountVectorizer`类，建议你花点时间了解下（第9章将会用到）。然而在分词方面，NLTK提供更多选择。如果你打算用Python做自然语言处理，NLTK是个不错的选择。

6.2.2　N 元语法

比起用单个词作特征，使用N*元语法*能更好地描述文档，具体优势稍后会讲。N元语法是指由几个连续的词组成的子序列。拿我们的数据集来讲，N元语法指的是每条消息里一组连续的词。

N元语法的计算方法跟计算单个词语方法相同，我们把构成N元语法的几个词看成是*词袋*中的一个词。数据集[1]中每一项就变成了N元语法在给定文档中的词频。

> *N元语法中的参数n，对于英语这门语言，一开始取2到5之间的值就可以，有些应用可能要使用更高的值*

举个例子吧，当n取3时，我们从下面引文中抽取前几个N元语法。

① 指用N元语法作为特征的数据集。——译者注

Always look on the bright side of life.

第一个N元语法（三元）是Always look on，第二个是look on the，第三个是on the bright。你可能已经发现，几个N元语法有重合，其中三个词有不同程度的重复。

N元语法比起单个词有很多优点。这个简单的概念不用通过大量的计算，就提供了有助于理解词语用法的上下文信息。它的缺点是特征矩阵变得更为稀疏——一个N元语法不太可能出现两次（尤其是在Twitter消息及其他短文本中！）。

对于社会媒体所产生的内容以及其他短文档，N元语法不可能出现在多篇不同的文档中，除非是转发。然而，在长文档中，N元语法就很有效。

文档的另外一种N元语法关注的不是一组词而是一组字符（虽然字符N元语法[①]有多种计算方法！）。字符N元语法有助于发现拼写错误，除此之外，还有其他好处。本章及第9章都将测试这种N元语法的效果。

6.2.3 其他特征

除了N元语法外，还可以抽取很多其他特征，其中就包括句法特征，比如特定词语在句子中的用法。对于需要理解文本含义的数据挖掘应用，往往会用到词性。本书限于篇幅将不涉及这些特征。如果你对此感兴趣，推荐你看由Packt出版的*Python 3 Text Processing with NLTK 3 Cookbook*，作者为Jacob Perkins。

6

6.3 朴素贝叶斯

毫无疑问，朴素贝叶斯概率模型是以对贝叶斯统计方法的朴素解释为基础。尽管存在朴素的一面，这种方法应用面很广且都取得了不错的效果。特征类型和形式多样的数据集也可以用它进行分类，本章重点讲解如何用二值化后的特征组成的词袋模型来分类。

6.3.1 贝叶斯定理

大多数人在开始学习统计学时，都会被灌输从频率论者角度出发看问题的思想，即假定数据遵从某种分布，我们的目标是确定该种分布的几个参数。我们假定参数是固定的（也可能不正确），然后用自己的模型来套这些数据，甚至通过测试来证明数据与我们的模型相吻合。

相反，贝叶斯统计实际上是根据普通人（非统计学家）实际的推理方式来建模。我们用拿到的数据，来更新模型对某事件即将发生的可能性的预测结果。在贝叶斯统计学中，我们使用

① 英文为Character N-gram。——译者注

数据来描述模型，而不是使用模型来描述数据，用数据证实拍脑瓜得出的模型是典型的频率论者的做法。

贝叶斯定理旨在计算P(A | B)的值，也就是在已知B发生的条件下，A发生的概率是多少。大多数情况下，B是被观察事件，比如“昨天下雨了”，A为预测结果“今天会下雨”。对数据挖掘来说，B通常是观察样本个体，A为被预测个体所属类别。下一节将学习如何在数据挖掘领域使用贝叶斯定理。

贝叶斯定理公式如下：

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

举例说明，我们想计算含有单词drugs的邮件为垃圾邮件的概率（正如我们认为含有该词的Twitter消息可能是兜售药品的垃圾广告）。

在这里，A为“这是封垃圾邮件”。[①]我们先来计算P(A)，它也被称为先验信念（prior belief）。计算方法是，统计训练集中垃圾邮件的比例。如果我们的数据集每100封邮件有30封垃圾邮件，P(A)为30/100或0.3。

B表示“该封邮件含有单词drugs”。类似地，我们可以通过计算数据集中含有单词drugs的邮件数量得到P(B)。如果每100封邮件中有10封邮件包含单词drugs，那么P(B)就为10/100或0.1。计算P(B)时，我们不关注邮件是不是垃圾邮件。

P(B|A)指的是垃圾邮件中含有单词drugs的概率，计算起来也很容易，统计训练集中所有垃圾邮件的数量以及其中含有单词drugs的数量。30封垃圾邮件中，如果有6封含有单词drugs，那么P(B|A)就为6/30或0.2。

现在，我们根据贝叶斯定理就能计算出P(A|B)，得到含有drugs的邮件为垃圾邮件的概率。把上面求出来的各项代入前面的贝叶斯公式，得到结果0.6。这表明如果邮件中含有drugs这个词，那么该邮件为垃圾邮件的概率为60%。

请注意上述例子的实证性特点——我们使用的经验或证据直接来自于数据集，而不是事先假定好的某种分布。相比之下，频率论方法会要求我们首先为训练集中的单词出现概率创建某种分布形式。

6.3.2　朴素贝叶斯算法

回过头看下贝叶斯公式，我们可以用它计算个体从属于给定类别的概率。因此，它可以用来

① 原文直译为“A为这是封垃圾邮件的概率”，其实此处A表示的是事件，而不是概率。——译者注

分类。

我们用C表示某种类别，用D表示数据集中一篇文档，来计算贝叶斯公式所要用到的各种统计量，对于不好计算的，做出朴素假设，简化计算。朴素贝叶斯分类算法使用贝叶斯定理计算个体从属于某一类别的概率。

P(C)为某一类别的概率，可以从训练集中计算得到（方法跟上文检测垃圾邮件例子所用到的一致）。统计训练集所有文档从属于给定类别的百分比。

P(D)为某一文档的概率，它牵扯到各种特征，计算起来很困难，但是在计算文档属于哪个类别时，对于所有类别来说，P(D)相同，因此根本就不用计算它。稍后我们来看下怎么处理。

P(D|C)为文档D属于C类的概率。由于D包含多个特征，计算起来可能很困难，这时朴素贝叶斯算法就派上用场了。我们朴素地假定各个特征之间是相互独立的，分别计算每个特征（D1、D2、D3等）在给定类别出现的概率，再求它们的积。

```
P(D|C) = P(D1|C) x P(D2|C).... x P(Dn|C)
```

上式右侧对于二值特征相对比较容易计算。直接在数据集中进行统计，就能得到所有特征的概率值。

相反，如果我们不做朴素的假设，就要计算每个类别不同特征之间的相关性。这些计算很难完成，如果没有大量的数据或足够的语言分析模型也不可能完成。

到这里，算法就很明确了。对于每个类别，我们都要计算P(C|D)，忽略P(D)项。概率较高的那个类别即为分类结果。由于P(D)对每个类别来说都是相等，去掉它对最终预测结果没有多大影响。

6.3.3 算法应用示例

举例说明下计算过程，假如数据集中有以下一条用二值特征表示的数据：[1，0, 0, 1]。

训练集中有75%的数据属于类别0，25%属于类别1，且每个特征属于每个类别的似然度如下。

类别0：[0.3，0.4，0.4，0.7]

类别1：[0.7，0.3，0.4，0.9]

拿类别0中特征1的似然度举例子，上面这两行数据可以这样理解：类别0中有30%的数据，特征1的值为1。

我们来计算一下这条数据属于类别0的概率。类别为0时，P(C=0) = 0.75。

朴素贝叶斯算法用不到P(D)，因此我们不用计算它。我们来看下计算过程。

```
P(D|C=0) = P(D1|C=0) x P(D2|C=0) x P(D3|C=0) x P(D4|C=0)
```

```
= 0.3 x 0.6 x 0.6 x 0.7
= 0.0756
```

> 第二、三个值为0.6，是因为在上面我们给出的那条数据（[1, 0, 0, 1]）中，这两个特征的值为0。而我们给出的似然度表示特征值取1时，在各类别的概率。因此，特征值为0的概率为：P(0) = 1 – P(1) 。

现在，我们就可以计算该条数据从属于每个类别的概率。需要提醒的是，我们没有计算P(D)，因此，计算结果不是实际的概率。由于两次都不计算P(D)，结果具有可比较性，能够区分出大小就足够了。来看下计算结果。

```
P(C=0|D) = P(C=0) P(D|C=0)
= 0.75 * 0.0756
= 0.0567
```

接着，计算类别1的概率。

```
P(C=1) = 0.25
```

朴素贝叶斯公式不需要计算P(D)。我们来看下计算过程。

```
P(D|C=1) = P(D1|C=1) x P(D2|C=1) x P(D3|C=1) x P(D4|C=1)
= 0.7 x 0.7 x 0.6 x 0.9
= 0.2646
P(C=1|D) = P(C=1)P(D|C=1)
= 0.25 * 0.2646
= 0.06615
```

> 通常，P(C=0|D) + P(C=1|D)应该等于1。毕竟，只有这两种选择！然而，我们这里二者的和不为1，因为我们在计算时省去了公式中的P(D)项。

该条数据应该被分到类别1中。计算过程中，你可能已经猜到结果了。看到最终结果两个类别的概率如此接近，你可能还是会有点惊讶。毕竟，类别为0时，P(D|C)的概率比类别为1时高很多。这是因为我们给出的先验概率很高，即大部分数据都属于类别0。

如果训练集中两类数据数量相同，结果就会大为不同。假设P(C=0) 、P(C=1) 各为0.5，再计算下结果看看。

6.4　应用

接下来，创建流水线，接收一条消息，仅根据消息内容，判断它是否与编程语言Python相关。

我们使用NLTK抽取特征。NLTK提供了大量用于自然语言处理的工具。后续章节还会继续使用它。

> 用pip安装NLTK：`pip3 install nltk`
> 如果安装失败，请参考NLTK安装指南：www.nltk.org/install.html。

接下来，创建流水线抽取词语特征，并使用朴素贝叶斯算法对消息进行分类。流水线包括以下步骤。

(1) 用NLTK的`word_tokenize`函数，将原始文档转换为由单词及其是否出现组成的字典。

(2) 用`scikit-learn`中的`DictVectorizer`转换器将字典转换为向量矩阵，这样朴素贝叶斯分类器就能使用第一步中抽取的特征。

(3) 正如前几章做过的那样，训练朴素贝叶斯分类器。

(4) 还需要新建一个笔记本文件ch6_classify_twitter（本章最后一个），用于分类。

6.4.1　抽取特征

我们使用NLTK抽取单词是否出现作为特征。我们想在流水线中使用NLTK，但是它的接口与转换器接口不一致。因此，需要创建一个包含`fit`和`transform`方法的基础转换器，只有这样才能在流水线中使用。

首先，创建转换器类。这个类不需要进行预处理，只需要抽取特征。因此，`fit`函数不做任何操作，只返回它自身（self）即可，转换器对象要用到它。

转换器函数就有点复杂。我们要用它从每篇文档中抽取单词，如果单词出现，记为`True`。注意这里使用二值特征，单词在文档中出现，值为`True`，反之，值为`False`。如果我们想使用词频，就要创建用来统计单词频率的字典，前几章讲过。

来看下代码。

```
from sklearn.base import TransformerMixin
class NLTKBOW(TransformerMixin):
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return [{word: True for word in word_tokenize(document)}
                 for document in X]
```

返回结果为一个元素为字典的列表，第一个字典的各项为第一条消息中的所有词语。字典的每一项用单词作为键，值为`True`，表示该词在该条消息中出现过。字典中没有出现的词，表示这条消息里不包含该词。当然，我们也可以用值`False`来表示没在消息中的词，但是那样太浪费

存储空间，也没有必要。

6.4.2　将字典转换为矩阵

这一步是将上面得到的字典转换为可以用分类器进行处理的矩阵。在DictVectorizer的帮助下，这一步变得非常容易。

DictVectorizer类接受元素为字典的列表，将其转换为矩阵。矩阵中的各个特征为所有字典中的每个键，特征值就是特征在文本中是否出现。用代码生成字典很容易，但是实现的很多数据挖掘算法更喜欢接收矩阵格式的数据，于是DictVectorizer显得格外有用。

数据集中，每个字典用单词作为键，单词只有在对应的消息中出现，这个单词才会出现在字典里。因此，矩阵以每个单词作为特征，如果消息中出现该单词，那么相应的特征值就为True。

导入DictVectorizer后，就可以使用它。

```
from sklearn.feature_extraction import DictVectorizer
```

6.4.3　训练朴素贝叶斯分类器

最后，我们需要组装分类器，因为本章使用朴素贝叶斯算法，数据集只包含二值特征，因此我们使用专门用于二值特征分类的BernoulliNB分类器，它用起来很简单。就像是DictVectorizer一样，我们先来导入它，把它添加到流水线中。

```
from sklearn.naive_bayes import BernoulliNB
```

6.4.4　组装起来

终于等到把所有部件组装起来的时候了。像前面做过的那样，在笔记本文件中，指定好文件名，加载数据集和类别数据。注意是消息（不是消息编号）及它们的类别所在的文件。代码如下：

```
import os
input_filename = os.path.join(os.path.expanduser("~"), "Data",
  "twitter", "python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data",
  "twitter", "python_classes.json")
```

加载消息。我们只对消息内容感兴趣，因此只提取和存储它们的text值。代码如下：

```
tweets = []
with open(input_filename) as inf:
```

```
    for line in inf:
        if len(line.strip()) == 0:
            continue
        tweets.append(json.loads(line)['text'])
```

加载消息的类别。

```
with open(classes_filename) as inf:
    labels = json.load(inf)
```

创建流水线，把所有部件组合起来。流水线包含以下三个部分。

- 我们创建的NLTKBOW转换器
- DictVectorizer转换器
- BernoulliNB分类器

流水线代码如下：

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([('bag-of-words', NLTKBOW()),
                     ('vectorizer', DictVectorizer()),
                     ('naive-bayes', BernoulliNB())
                     ])
```


我们几乎现在就可以运行流水线，用之前多次用过的cross_val_score方法来计算正确率。但是在这之前，我们要介绍一种比正确率更好的评价指标。我们后面会看到，每个类别数据量不同的情况下，正确率不足以说明算法的优劣。

6.4.5 用 F1 值评估

选择评价指标时，了解它们的适用范围很重要。正确率应用范围很广，理解起来比较容易，计算起来也方便。但是，造假很容易。换句话说，你很容易就能实现一个正确率很高，但实际用处不大的算法。

我们的消息数据集（你的可能与此不同）中，50%的消息与编程语言有关，50%不相关，很多数据集不会这么均匀（balanced）。

例如，对于垃圾邮件过滤器而言，其所处理的邮件很可能80%以上都是垃圾邮件，倘若一个过滤器把所有邮件都标为垃圾邮件，它没有实际应用价值，但是正确率却高达80%！

为了解决这个问题，我们使用另一个最为常用的评价指标F1值（也被称为F值、F-measure，或者其他变体[1]）。

[1] 比如F0.5等。——译者注

F1值是以每个类别为基础进行定义的，包括两大概念：准确率（precision）和召回率（recall）。准确率是指预测结果属于某一类的个体，实际属于该类的比例。召回率是指被正确预测为某个类别的个体数量与数据集中该类别个体总量的比例。

在我们这里，可分别对两个类别（相关和不相关）的分类情况计算F1值。但是，我们只关注相关这一类数据的分类情况。因此，准确率计算变为以下问题：在所有被预测为相关的消息中，真正相关的占比多少？类似地，召回率就转化为：数据集所有相关的消息中，有多少被正确预测为相关的？

计算出准确率和召回率后，就能得到F1值，它是两者的调和平均数。

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

将`scoring`参数的值设置为F1，就能使用`scikit-learn`中的F1方法。默认将会返回类别为1的F1值。使用以下代码求得F1值。

```
scores = cross_val_score(pipeline, tweets, labels, scoring='f1')
```

输出平均值。

```
import numpy as np
print("Score: {:.3f}".format(np.mean(scores)))
```

结果为0.798，这表明预测含有Python的消息与编程语言有关的F1值差不多为80%。这是使用包含200条消息的数据集取得的结果。如果采集更多消息作为训练数据，你会发现结果还会提升！

更多的数据通常意味着更好的结果，但也不一定！

6.4.6　从模型中获取更多有用的特征

你可能存在这样的疑问：“什么特征才是判断一条消息是否相关的最好的特征？”我们可以从朴素贝叶斯模型中抽取该信息，找到朴素贝叶斯算法所认为的最好的特征。

首先，训练得到一个新模型。使用`cross_val_score`在测试集上得到交叉检验结果不难，但是要得到模型本身就不容易了。为了得到模型，我们只好用流水线的`fit`函数，创建新模型。代码如下：

```
model = pipeline.fit(tweets, labels)
```

> 注意我们不是在评价模型，训练/测试集的切分就没那么严格。然而，在使用这些特征之前，应该在单独的测试集上评价其表现。为保证讲解得清楚明了，跳过这一部分。

借助流水线的named_steps属性和步骤名（创建流水线对象时，我们自己定义的），就能访问流水线每一个步骤。例如，可以像下面这样访问朴素贝叶斯模型。

```
nb = model.named_steps['naive-bayes']
```

从这个模型中，可以抽取每个单词的对数概率，比如log(P(A|f))，其中f为给定特征。

之所以使用对数概率，是因为实际值非常小。例如，第一个值为–3.486，实际概率约为0.03。计算中涉及很小的概率值时，常使用对数概率来防止数值下溢，因为非常小的值往往会被约等于0。所有概率连乘，其中一个值为0，最终结果将为0！而实际上即使是很小的值，大小不同，对分类的贡献率也会有所差异，值越大的特征贡献率越大。

把得到的对数概率数组按照降序排列，找出最有用的特征。降序排列，需要在值前加个负号。代码如下：


```
top_features = np.argsort(-feature_probabilities[1])[:50]
```

上面代码只是给出特征索引值而没有给出实际的特征名称。这样看不出什么东西来，因此，需要把特征索引和特征名称对应起来。流水线的DictVectorizer这一步很关键，它是用来创建矩阵的。幸运的是，它也记录了特征名称和索引值的映射关系。因此，可以从流水线的这一部分抽取特征。

```
dv = model.named_steps['vectorizer']
```

通过在DictVectorizer的feature_names_属性中进行查找，找到最佳特征的名称，然后将其输出。输入下面代码，运行后将得到最佳特征列表。

```
for i, feature_index in enumerate(top_features):
    print(i, dv.feature_names_[feature_index],
      np.exp(feature_probabilities[1][feature_index]))
```

前几个特征为“:”“http”“#”“@”。结合采集的数据来判断，我们认为这些很可能是噪音（虽然冒号在编程语言之外的文本中使用的相对较少）。更多的训练数据，有助于减少这些噪音对分类的影响。接着往下看，下面这些特征更像是与编程语言有关。

```
7 for 0.188679245283
11 with 0.141509433962
28 installing 0.0660377358491
29 Top 0.0660377358491
34 Developer 0.0566037735849
```

```
35 library 0.0566037735849
36 ] 0.0566037735849
37 [ 0.0566037735849
41 version 0.0471698113208
43 error 0.0471698113208
```

还有一些特征是在工作环境中提及Python，因此可能指的是编程语言Python。（虽然作为自由职业者的耍蛇人也可能使用Python这个词，但是他们较少活跃在Twitter上。）

```
22 jobs 0.0660377358491
30 looking 0.0566037735849
31 Job 0.0566037735849
34 Developer 0.0566037735849
38 Freelancer 0.0471698113208
40 projects 0.0471698113208
47 We're 0.0471698113208
```

上面最后一个特征通常出现在诸如“We’re looking for a candidate for this job”（我们正在招聘符合该职位要求的员工）这样的消息中。

查看这些特征，我们收获不小，有了这些特征，我们自己经过训练也可以完成分类任务，寻找这些特征的共同点（与主题相关），或者去除讲不通的特征。例如，词“RT”虽然排名比较靠前，然而，这是Twitter网站表示转发的用语。经验丰富的工作人员就可以从特征列表中删除这个词，降低由于数据集很小而引入的噪音对分类结果的影响。

6.5　小结

本章研究的是文本挖掘——特征的抽取、应用及扩展方法，具体研究任务是根据语境消除词语的歧义——即一条消息中的Python是否指编程语言。我们使用Twitter提供的API下载消息，使用在笔记本文件中建立的表单完成语料标注。

我们还考虑了实验结果的可再现性。虽然Twitter不允许你把自己采集的数据给别人使用，但是消息编号是可以共享的。我们编写代码，保存消息编号，再用这些编号来重建先前的数据集。由于有些消息被删除或是出于其他原因，我们无法再次获取它们。

我们用朴素贝叶斯分类器对文本进行分类，该分类器是以贝叶斯定理为基础，它使用数据来更新模型，而不是像频率论者那样从模型出发。这有助于整合新数据到模型中，利用新数据来更新模型和使用先验知识。此外，基于各特征相互独立的朴素假设，便于计算各特征出现在给定类别的概率，而不用考虑各特征之间复杂的相关性。

我们用词语是否出现作为特征值——即消息是否含有某个词，这种模型[①]叫作词袋模型。虽

① 把文档看成由一个个孤立的、无前后位置关系的单词组成的。——译者注

然它丢掉了词语在句子中的位置等信息，但它在很多数据集上表现不凡。

朴素贝叶斯分类器结合词袋模型，组装成的流水线功能强大。对于大多数文本挖掘任务，它都能取得很好的效果。在尝试更高级的模型之前，用这样的分类器取得的分类结果作为参考的基准很不错。另外一个优点是，朴素贝叶斯分类器不需要调整任何参数（如果你愿意折腾的话，也确实有几个）。

下一章将介绍怎样从另外一种数据类型——图中抽取特征，尝试解决向社会媒体网站用户推荐感兴趣的人这一问题。

第 7 章

用图挖掘找到感兴趣的人

很多事物都可用图来表示，大数据、社交网络和物联网时代尤其如此。特别值得一提的是，社交网络具有很大的商业价值，像Facebook这样每天有5亿活跃用户（50%的用户每天都会登录）的网站，通过定向投放广告获利颇丰。怎样才能打造一个这样的网站呢？对于网站而言，要想留住用户，你得有用户感兴趣的人或内容。

本章将探讨相似度这个概念以及如何根据它来创建图模型，并且探讨如何使用连通分支（connected components）把大图分为有意义的小图。本章即将介绍的算法引入聚类分析概念——根据相似度，把大数据集划分为几个子集。第10章将对聚类分析做更加深入的讲解。

本章主要内容如下：

- 用社交网络数据创建图
- 加载和保存创建的分类器
- NetworkX库
- 图到矩阵的转换
- 距离和相似度
- 根据打分函数优化参数
- 损失函数和打分函数

7.1 加载数据集

本章的任务是根据社交网络用户的好友信息，向他们推荐好友。逻辑为：如果两个用户有共同的好友，那么这两个人相似度很高，值得向彼此推荐。

利用上一章介绍的Twitter API来获取数据，创建一个小的社交网络图。寻找喜欢同一个话题（与上一章相同，依旧是编程语言Python）的用户，从中选一部分，再获取这些人的好友列表（他们关注的人）。有了以上数据，就能根据两个用户共同拥有的好友数量，计算他们的相似度。

> 除了Twitter，还有很多其他社交网站。之所以使用Twitter，是因为其提供的API很容易获取所需要的数据。其他网站，比如Facebook、LinkedIn和Instagram等也开放这些数据，但是获取起来难度更大些。

现在来采集数据，新建一个IPython Notebook笔记本文件，初始化`twitter`连接实例，方法与上一章相同。可以沿用上一章所创建的Twitter应用的密钥信息，再创建新的也可以：

```
import twitter
consumer_key = "<Your Consumer Key Here>"
consumer_secret = "<Your Consumer Secret Here>"
access_token = "<Your Access Token Here>"
access_token_secret = "<Your Access Token Secret Here>"
authorization = twitter.OAuth(access_token, access_token_secret,
  consumer_key, consumer_secret)
t = twitter.Twitter(auth=authorization, retry=True)
```

指定输出文件名：

```
import os
data_folder = os.path.join(os.path.expanduser("~"), "Data",
  "twitter")
output_filename = os.path.join(data_folder, "python_tweets.json")
```

依然用`json`库保存数据：

```
import json
```


接下来，收集一组用户信息。像上一章那样搜索推文，查找包含单词python的消息。首先，创建两个列表，用于存储消息文本内容及相应的用户信息。稍后会用到用户编号，因此，需要创建个字典，将用户昵称和编号对应。代码如下：

```
original_users = []
tweets = []
user_ids = {}
```

搜索包含python的消息，遍历搜索结果：

```
search_results = t.search.tweets(q="python",
  count=100)['statuses']
for tweet in search_results:
```

我们只对消息感兴趣，对Twitter可能返回的其他信息不感兴趣，因此检测消息中有没有text属性：

```
    if 'text' in tweet:
```

如果有，记录用户昵称、消息正文，并且将用户昵称和编号关联起来。代码如下：

```
        original_users.append(tweet['user']['screen_name'])
        user_ids[tweet['user']['screen_name']] =
          tweet['user']['id']
        tweets.append(tweet['text'])
```

运行上述代码将会得到大约100条消息，有时可能会少些。但是，这些消息并不是全都和编程语言有关。

7.1.1　用现有模型进行分类

正如上一章所说，不是所有提到python的消息都与编程语言有关。怎么才能找出所需要的消息？上一章用过的分类器就能派上用场了。分类器虽不完美，但使用它进行分类，能够过滤掉搜索结果中相当一部分噪音。

在本案例中，只对谈论Python编程语言的用户感兴趣。使用上一章创建的分类器找出与编程语言Python相关的消息。发表这类消息的人是我们真正感兴趣的用户。具体做法如下。

首先，需要保存朴素贝叶斯分类模型。打开上一章创建分类器的IPython笔记本文件。如果已经关闭，笔记本不记得之前的操作，需要运行所有的代码片段，方法是点击笔记本的Cell菜单，选择Run All。

运行完后，选择页面下方的空白格子。如果没有，选中最后一个格子，点击Insert菜单，选择Insert Cell Below选项。

我们将使用`joblib`库保存并加载模型。

> `joblib`包含在`scikit-learn`库中。

首先，导入`joblib`库，为模型指定输出文件名（确保目录存在，否则无法创建）。我把模型保存在我自己创建的Models目录下，你也可以将其保存到其他地方。代码如下：

```
from sklearn.externals import joblib
output_filename = os.path.join(os.path.expanduser("~"), "Models",
  "twitter", "python_context.pkl")
```

接着，用`joblib`库的`dump`函数，与`json`库的`dump`函数功能类似。参数为模型本身（怕你忘了，捎带提下，模型名称为`model`）和输出文件名。

```
joblib.dump(model, output_filename)
```

运行上述代码，模型将会保存到指定文件中。接着，回到上小节创建的笔记本文件，加载模型。

复制下面代码，在当前笔记本文件中再次指定模型的文件名：

```
model_filename = os.path.join(os.path.expanduser("~"), "Models",
  "twitter", "python_context.pkl")
```

检查下这里的文件名是否与保存模型所用的文件名相同。接着，需要重建NLTKBOW类，因为它是定制的类，无法直接用joblib加载。更好的处理方法后续章节会讲。现在，只需从上一章的代码中复制整个NLTKBOW类及它所依赖的模块：

```
from sklearn.base import TransformerMixin
from nltk import word_tokenize

class NLTKBOW(TransformerMixin):
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        return [{word: True for word in word_tokenize(document)}
                 for document in X]
```

现在，只需调用joblib的load函数就能加载模型：

```
from sklearn.externals import joblib
context_classifier = joblib.load(model_filename)
```

context_classifier与第6章的model对象功能相同，是Pipeline对象的实例，它的三个步骤也与第6章流水线（NLTKBOW、DictVectorizer和BernoulliNB分类器）相同。

调用context_classifier模型的predict函数，预测消息是否与编程语言相关。代码如下：

```
y_pred = context_classifier.predict(tweets)
```

如果第i条消息与编程语言有关，那么y_pred中的第i项为1，否则为0。据此，可以获取到所有与编程语言有关的消息及用户：

```
relevant_tweets = [tweets[i] for i in range(len(tweets)) if y_pred[i]
  == 1]
relevant_users = [original_users[i] for i in range(len(tweets)) if
  y_pred[i] == 1]
```

用我采集的数据，共找到46名相关用户。比起之前的100条消息/用户显得有点少，但还是可以以此为基础来构建社交网络。

7.1.2 获取 Twitter 好友信息

接下来，需要获得以上每个用户的好友。这里好友的定义是：用户正在关注的人。Twitter提供了相应API：friends/ids，它好用的地方在于一次调用最多能获得高达5000个好友的编号，

不好的地方是每15分钟最多只能调用15次，这也就意味着获取每个用户的好友列表至少需要一分钟——如果好友数量多于5000，需要更多时间（这种情况比想象中的更常见）。

然而，代码相对比较简单。接下来两节还要用到获取好友列表的功能，干脆把它封装成一个函数。首先，声明该函数，接收两个参数，一个是用于连接Twitter的对象，另一个是用户编号。函数返回由第二个参数指定的用户的所有好友，创建列表`friends`来存储好友们的编号。由于要用到时间模块`time`，顺便导入它。在给出完整的函数前，会逐一讲解每一部分代码。开始部分如下：

```
import time
def get_friends(t, user_id):
    friends = []
```

听起来有些奇怪，很多Twitter用户拥有5000个以上的好友。因此，要用到Twitter的pagination（分页）对象。Twitter使用游标管理多页数据。当向Twitter请求数据时，不仅返回所需数据，还返回数据类型为整数的游标，Twitter用游标跟踪每一次请求。如果没有更多内容，游标为0；否则，可以使用游标获得下一页数据。开始把游标设置为–1，表明是数据的开始：

```
    cursor = -1
```

接下来，只要游标不为0（当为0时，数据已采集完），就持续执行下面的循环。请求用户的好友数据，并将这些好友的编号添加到`friends`列表中。这里用到了`try`语句，请求过程中可能会遇到问题，使用`try`以便于处理异常。`results`字典中键为`ids`的那一项保存的是好友编号，将好友编号添加到`friends`列表中。然后，更新游标。下一次迭代时，游标使用刚更新过的值。最后，检查好友数量是否超过一万，如果超过，跳出循环。代码如下：

```
    while cursor != 0:
        try:
            results = t.friends.ids(user_id= user_id,
              cursor=cursor, count=5000)
            friends.extend([friend for friend in results['ids']])
            cursor = results['next_cursor']
            if len(friends) >= 10000:
                break
```

> 这里有必要插入条警告信息。从互联网抓取数据，可能会时不时遇到这样那样奇怪的事情。我在编写上面这段代码时遇到的一个问题是有些用户好友异常多。为了应对这个问题，代码中增加了安全退出机制，用户好友再多，只要获取到他的一万名好友信息后就强制退出。如果想采集用户的所有好友，请删掉这两行，但是请注意碰巧遇到好友特别多的用户，会卡很长时间。

接着为可能捕获到的异常编写处理方法。最可能犯的错误是不小心达到了API访问次数的上限（虽然sleep语句可以在一定程度上避免这种情况，但在sleep语句结束前，中止代码执行后，接着在原本应该停顿的时间重新运行代码，就会出现这种情况）。这样的话，返回结果results为None，代码会抛出TypeError（类型错误）异常。遇到这种情况，等待五分钟，再次执行新一轮循环，希望Twitter能够重新计算时间，这样在接下来的15分钟就又能请求15次数据。除此之外，可能还有其他原因引发的TypeError异常。对于这种异常，需要单独处理，因此需要返回具体错误信息，方便查找错误起因。代码如下：

```
        except TypeError as e:
            if results is None:
                print("You probably reached your API limit,
                  waiting for 5 minutes")
                sys.stdout.flush()
                time.sleep(5*60) # 5 minute wait
            else:
                raise e
```

第二类异常可能发生在Twitter这一端，比如查找的用户不存在或是其他和数据相关的错误。这时，不要再尝试继续获取导致报错的用户的好友数据，返回已经得到的即可（很可能为0）。代码如下：

```
        except twitter.TwitterHTTPError as e:
            break
```

接着处理API限制。Twitter只允许每15分钟调用15次获取用户好友数据的函数，15次过后，需要等一分钟再继续进行。把这段代码放到finally语句中，遇到异常后，也会执行：


```
        finally:
            time.sleep(60)
```

函数最后返回所采集到的好友编号：

```
    return friends
```

下面是完整的函数：

```
import time
def get_friends(t, user_id):
    friends = []
    cursor = -1
    while cursor != 0:
        try:
            results = t.friends.ids(user_id= user_id,
              cursor=cursor, count=5000)
            friends.extend([friend for friend in
              results['ids']])
            cursor = results['next_cursor']
            if len(friends) >= 10000:
                break
```

```
            except TypeError as e:
                if results is None:
                    print("You probably reached your API limit,
                      waiting for 5 minutes")
                    sys.stdout.flush()
                    time.sleep(5*60) # 5 minute wait
                else:
                    raise e
            except twitter.TwitterHTTPError as e:
                    break
                   finally:
                       time.sleep(60)
        return friends
```

7.1.3　构建网络

现在着手构建网络。从最初的46名（你得到的数据集可能与我的有差异）用户出发，找到每个人的好友，将其保存到字典里（从`user_id`字典拿到用户编号）：

```
friends = {}
for screen_name in relevant_users:
    user_id = user_ids[screen_name]
    friends[user_id] = get_friends(t, user_id)
```

删除没有好友的孤家寡人。对于这些人，无法按照既定逻辑向他们推荐好友。也许可以从他们发表的消息以及关注他们的人那里发现有用线索。感兴趣的读者可自行研究，限于篇幅，本章不过多涉及。把这些人过滤掉。代码如下：

```
friends = {user_id:friends[user_id] for user_id in friends
             if len(friends[user_id]) > 0}
```

这样能剩下30到50个用户，具体数量与你之前的搜索结果有关。这些不大够用，得想办法再抓取些数据，使总用户增加到150个。下面的代码运行起来要费点时间——由于Twitter API的限制，一分钟只能取到一个用户的好友数据。不用算也知道150个用户数据，需要150分钟，也就是2.5个小时。既然要花这么多时间来采集用户的好友数据，得保证这些用户是想要的才行。

那么，什么用户才是想要的呢？考虑到要根据共同的好友向用户推荐他们可能感兴趣的人，因此就查找有共同好友的人。找到现有用户的好友，从他们中间找出在现有用户中间有更多好友的人。因此，需要统计这些用户的好友数量，即出现在已有用户的`friends`列表中的次数。从事数据挖掘任务时，确定采样策略前，考虑好应用的实际目标很有必要。这里向相似的用户推荐好友更可行。

具体做法为遍历所有用户的好友列表，统计每个好友的出现次数：

```
from collections import defaultdict
def count_friends(friends):
    friend_count = defaultdict(int)
```

```
    for friend_list in friends.values():
        for friend in friend_list:
            friend_count[friend] += 1
    return friend_count
```

计算完成后，根据好友数量对friend_count字典进行排序，找到在当前用户中，关系网最大、最密集的人。代码如下：

```
friend_count
reverse=True) = count_friends(friends)
from operator import itemgetter
best_friends = sorted(friend_count.items(), key=itemgetter(1),
```

建立一循环，凑够150个用户的好友数据后，该循环就会结束。遍历best_friends字典（按照在现有用户中的好友数多少排序），找到还没有获取好友列表的用户，然后获取他的好友列表，更新friends列表。最后，再次查找谁是最受欢迎的①：

```
while len(friends) < 150:
    for user_id, count in best_friends:
        if user_id not in friends:
            break
        friends[user_id] = get_friends(t, user_id)
    for friend in friends[user_id]:
        friend_count[friend] += 1
    best_friends = sorted(friend_count.items(),
      key=itemgetter(1), reverse=True)
```

上述代码运行结束后，就可以得到150名用户的好友数据。

> 你可能想把最大循环数设置得小一点，比如40或50（甚至可以先跳过这部分代码）。先敲完本章剩余代码，感觉下效果如何。再把循环数设置为150，等待几个小时，再来看一下结果。

鉴于上述采集数据的程序大约要运行两个小时，最好把中间结果保存下来，因为有时不得不关闭计算机。使用json库，就可以轻松把好友字典保存到文件里：

```
import json
friends_filename = os.path.join(data_folder, "python_friends.json")
with open(friends_filename, 'w') as outf:
    json.dump(friends, outf)
```

使用json.load函数，从文件中加载数据：

```
with open(friends_filename) as inf:
    friends = json.load(inf)
```

① friend_count字典更新后，刚刚找到的好友信息已经被包括进来了。——译者注

7.1.4　创建图

到现在为止，已经拿到了用户列表和他们的好友列表,很多用户是其他用户的好友。用这些存在好友关系的数据就能构建一张图（虽然这里好友关系不必是双向的）。

图由一组顶点和边组成。顶点通常表示对象——在这个例子中，顶点代表用户。边表示*用户A是用户B的好友*。由于顶点的顺序是有特殊含义的，该图称为有向图，因为用户A是用户B的好友并不代表用户B是用户A的好友[1]。可以用`NetworkX`库实现图关系的可视化。

> 可以用pip安装`NetworkX`库：`pip3 install networkx`。

首先，使用`NetworkX`创建有向图。根据习惯，导入`NetworkX`后给它一个简称`nx`（虽然不是必须这么做）。代码如下：

```
import networkx as nx
G = nx.DiGraph()
```

我们只将150名核心用户彼此间的好友关系绘制成图像[2]，其他好友关系由于数据量很大难以可视化（成千上万个，作图有难度）。把核心用户作为顶点，添加到图中。代码如下：

```
main_users = friends.keys()
G.add_nodes_from(main_users)
```

接着要创建边。如果第二个用户是第一个用户的好友，那么就在这两个顶点之间建立一条边。遍历所有的核心用户：

```
for user_id in friends:
    for friend in friends[user_id]:
```

确保好友在核心用户之列（当下先不关注非核心用户），然后为两者之间添加边。代码如下：

```
        if friend in main_users:
          G.add_edge(user_id, friend)
```

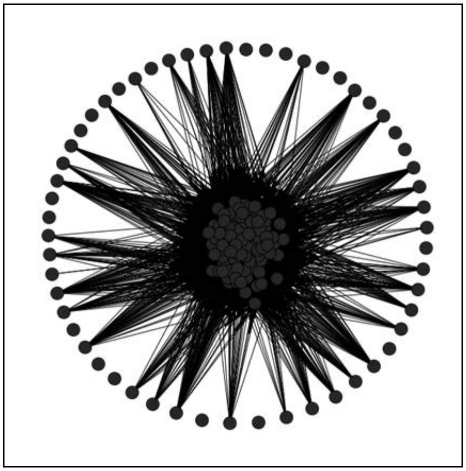
用`NetworkX`的`draw`函数为创建好的图绘制图像，`draw`函数要用到`matplotlib`库。在当前笔记本文件中作图，需要使用`inline`函数。代码如下：

```
%matplotlib inline
nx.draw(G)
```

① 这里的“好友”译成“关注的人”更好理解，表示一种单向关系。其实在现实生活中，也存在这种“单相思”的情况，你把他人当成是最好的朋友，人家未必如此。——译者注

② 注意这里“图像”和“图”的区别，前者指的是绘画、可视化意义上点、线条等形状的组合。后者是一种抽象的数据结构。文中“作图”表示绘制图像。——译者注

生成的图像理解起来有点困难；有几个顶点之间边较少，但大多数顶点之间边较多。

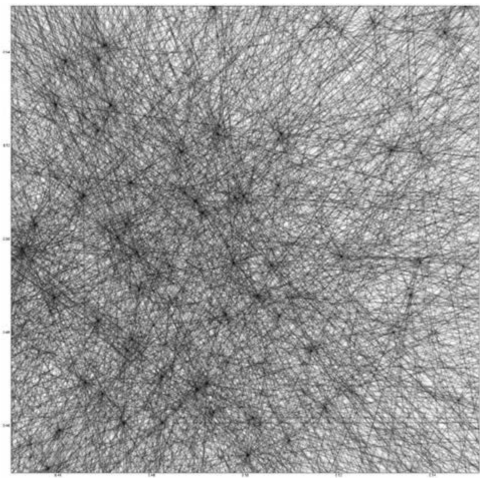


可以借助用于处理图像生成的pyplot函数设置图像大小。导入pyplot，把图像尺寸设置得大一点，调用NetworkX的draw函数（NetworkX使用pyplot函数作图）：

```
from matplotlib import pyplot as plt
plt.figure(3,figsize=(20,20))
nx.draw(G, alpha=0.1, edge_color='b')
```

生成的图像太大一页放不下，但是大图像的好处是：图的轮廓看得更清楚。在我创建的图中，一部分用户互为好友，但是其他绝大部分用户之间不存在好友关系。我将上述代码中的alpha设置得很小，这样边的颜色就为蓝色，将上图放大后从中间位置截取了下面这张图。

如你所见，中间位置各顶点之间连接有多紧密！



这与我们选择新用户的方法有关——选择那些已经在图中有着丰富连接的用户，因此他们可能只会把这个圈子扩展得更大些。一般而言，一个社交网络用户所拥有的连接数遵循幂定律。只有少数用户拥有很多连接数，大部分用户只有很少的连接数，描述这种关系的函数图像呈现出长尾形状。这里的数据集不遵从该幂定律，因为后来的核心用户产生于已有核心用户的共同好友。

7.1.5　创建用户相似度图

本章的任务是向拥有共同好友的用户推荐彼此。再次重申，逻辑是：*如果两个用户有共同的好友，那么这两个用户高度相似*。那么，可以向他们推荐彼此。

在现有图（边表示关注关系）基础上创建一个新图。新图中，顶点仍然是用户，边要升级为带权重的。带权重的边指的是边有了权重属性。权重越大表明两顶点相似度越高。但是，权重与应用场景有关。如果权重代表距离，那么权重越低表示相似度越高。

对于这个应用，边的权重表示由该条边所连接的两个用户之间的相似度（以共同好友数量为基础）。新图中的边没有方向性，因为A和B之间的相似度等于B和A之间的相似度。

计算两个列表之间的相似度有多种方法。就拿我们这里的两个用户来说，可以简单统计他们好友列表中共同好友数量。然而，这样做有个问题：好友更多的用户，共同好友数量通常也更多。于是，可以再除以他们拥有的不同好友的数量实现数据的规范化，得到的正是*杰卡德相似系数*（Jaccard Similarity）。

杰卡德相似系数总是在0到1之间，代表两者重合的比例。正如第2章讲到的，规范化是数据挖掘的一个重要方法，要坚持使用（除非有充分理由不这样做）。

在计算杰卡德相似系数时，需要用两个集合（好友数据）交集的元素数量除以两个集合并集的元素数量。这里要用到集合操作，但是好友数据存储在`friends`列表中，所以首先要把列表转换为集合。代码如下：

```
friends = {user: set(friends[user]) for user in friends}
```

创建一函数[①]，计算两个好友集合之间的相似度：

```
def compute_similarity(friends1, friends2):
return len(friends1 & friends2) / len(friends1 | friends2)
```

现在就可以创建加权图。本章后面会多次用到创建图的功能，因此将其封装成函数，留意下`threshold`参数：

```
def create_graph(followers, threshold=0):
    G = nx.Graph()
```

① 注意函数体`return`语句需要缩进。——译者注

使用嵌套循环遍历用户数据，跳过两层循环中用户相同的情况：

```
    for user1 in friends.keys():
        for user2 in friends.keys():
            if user1 == user2:
                continue
```

计算两个用户之间边的权重：

```
                weight = compute_similarity(friends[user1],
    friends[user2])
```

只有边的权重大于阈值，才会保存这条边，以确保只保存我们认为重要的边——例如，权重为0的边就没意义。阈值默认为0，因此会添加所有的边。本章稍后再指定一个合理的阈值，先这么用着吧。代码如下：

```
            if weight >= threshold:
```

如果权重大于等于阈值，把两个用户添加到图中（如果已经存在，不要重复添加）：

```
                G.add_node(user1)
                G.add_node(user2)
```

然后为这两个用户之间添加边，把权重设置为刚计算出来的相似度：

```
                G.add_edge(user1, user2, weight=weight)
```

循环结束后，图创建完毕，返回该图：

```
    return G
```

调用上述函数就能生成一个图。由于没有设置阈值，即使权重为0的边，也被创建出来。代码如下：

```
G = create_graph(friends)
```

我们创建的图各顶点之间连接得很紧密——所有顶点之间都有边，虽然很多边的权重为0。在作图时，可以根据边的权重来指定所用线条的粗细——粗线表示权重大。

由于顶点数量较多，作图时考虑适当增加图像尺寸，顶点之间的边看上去会更加清楚：

```
plt.figure(figsize=(10,10))
```

在绘制带权重的边之前，需要先画出顶点。NetworkX根据特定标准用布局信息来决定顶点和边的位置。绘制网络图很困难，特别是顶点数量较多时，虽然有很多现成的布局方式，但是否好用很大程度上取决于你的数据集、个人喜好及作图目的。我发现spring_layout非常好用，除此之外，还有circular_layout（如果其他方式不好用，还可以试试这个）、random_layout、shell_layout和spectral_layout。

> 访问http://networkx.lanl.gov/reference/drawing.html，了解更多NetworkX布局方法。虽然用draw_graphviz选项使复杂程度有所增加，但效果很好，如果追求更好的视觉效果，值得研究一下，可考虑将其用到实际工作中。

使用spring_layout布局方法：

```
pos = nx.spring_layout(G)
```

使用pos布局方法，确定顶点位置：

```
nx.draw_networkx_nodes(G, pos)
```

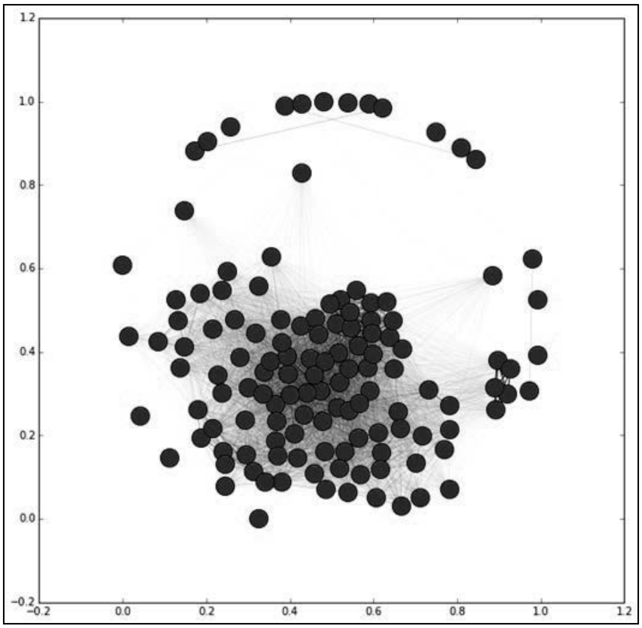
接下来，绘制边。遍历图中的每条边，获得其权重：

```
edgewidth = [ d['weight'] for (u,v,d) in G.edges(data=True)]
```

绘制各条边：

```
nx.draw_networkx_edges(G, pos, width=edgewidth)
```

最终图像具体长什么样取决于你的数据，但大体形状应该与我的一致：图中大量顶点之间有较多连接，少数顶点与网络其他顶点之间连接较少。



与本章前一个图的区别在于：该图顶点之间的边表示的是相似度而不再是好友关系（虽然这两者有相似之处）。可以从当前这个图中抽取信息用于好友推荐。

7.2 寻找子图

利用相似度函数求出每个用户与其他用户之间的相似度，根据相似度进行排序，找出与当前用户最相似的，把他推荐给当前用户——与推荐产品时的做法相同。然而，我们也可以找出批量相似度很大的用户。可以建议将这些用户组成一个群，向他们定向投放广告，或者即使只是向他们推荐群内的其他成员做好友。

找出这些相似用户群的任务叫作聚类分析。它的复杂程度一般分类算法比不了，可以说具有一定难度。例如，评价分类结果相对比较容易——比较使用分类器得到的结果与实际结果（训练集）就能知道正确率。但是聚类分析缺乏这样一个事实标准，评价结果时，只好根据经验来看分簇结果是否合乎情理。聚类分析另一个复杂之处在于，不能用事先标注好的数据进行训练——只好在使用聚类数学模型的基础上，求得近似的分组结果，而不是按照用户所期望的那样将数据分为一个个明确的类别。

7.2.1 连通分支

一种最简单的聚类方法就是找到图中的*连通分支*。一个连通分支是图中由边连接在一起的一组顶点，不要求顶点之间必须两两连接。但是，连通分支的任意两个顶点之间，至少存在一条路径。

> 连通分支计算时不考虑边的权重；只检查边是否存在。因此，下面代码将删除权重过低的边。[①]

NetworkX提供用于计算连通分支的函数，在图上调用即可。首先，用前面定义的create_graph函数创建一个新图，但这次指定阈值为0.1，只保留权重至少为0.1的边。

```
G = create_graph(friends, 0.1)
```

接着使用NetworkX提供的函数寻找图中的连通分支：

```
sub_graphs = nx.connected_component_subgraphs(G)
```

为了了解连通分支到底有多大，遍历找到的连通分支，输出其基本信息：

```
for i, sub_graph in enumerate(sub_graphs):
    n_nodes = len(sub_graph.nodes())
    print("Subgraph {0} has {1} nodes".format(i, n_nodes))
```

从输出结果，就能了解到每个连通分支有多大。在本例中，大的子图有62个用户，很多小子

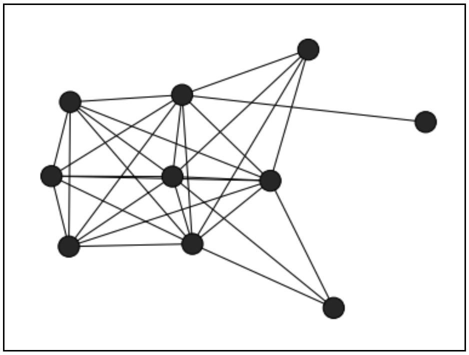
① 因为不论权重高低，只要有边连接的顶点就会被算到连通分支里，而一个连通分支被看作是一簇具有相似特点的用户，为了保证簇内用户具有较高的相似度，需要事先把权重低的边过滤掉。——译者注

图只有十几个甚至更少的用户。

可以通过调整阈值，找出不同的连通分支。这是因为阈值越高，顶点之间的边越少，因此连通分支规模更小，数量更多。设置更高的阈值，看看效果：

```
G = create_graph(friends, 0.25)
sub_graphs = nx.connected_component_subgraphs(G)
for i, sub_graph in enumerate(sub_graphs):
    n_nodes = len(sub_graph.nodes())
    print("Subgraph {0} has {1} nodes".format(i, n_nodes))
```

上述代码找到的连通分支数量多，每个连通分支所包含的顶点数量少。之前那个最大的子图至少被拆为三个部分，每部分都不超过10个用户。下图为其中一个连通分支，边也画出来了。因为这是一个连通分支，所以它里面的顶点跟大图中其他顶点之间不存在边（阈值至少为0.25）。



可以用不同的颜色把所有连通分支都画出来。因为各连通分支之间没有连接，因此没必要把它们画到一张图中。顶点和连通分支没有确定位置，把它们画到一张图上，反而会令人产生误解，以为它们之间的位置关系是确定的。鉴于此，我们可以把它们画到不同的图上。

在新单元格中，获得连通分支及它们的总数量。

```
sub_graphs = nx.connected_component_subgraphs(G)
n_subgraphs = nx.number_connected_components(G)
```

> 上述代码中，sub_graphs是生成器而不是连通分支列表。因此，需要用nx.number_connected_components找出连通分支的总数；由于NetworkX存储信息的方式比较特别，无法使用len函数。这正是重新计算连通分支的原因所在。

要显示所有的连通分支，图像得足够大。在用pyplot绘制图像时，让图像大小随着连通分支数量的增加而增加：

```
fig = plt.figure(figsize=(20, (n_subgraphs * 3)))
```

接下来，遍历所有的连通分支，为每一个连通分支作图。add_subplot的几个参数分别为图的行数、图的列数及图所在位置。我作图时使用了三列，你可以尝试其他值（记得同时修改行数和列数）：

```
for i, sub_graph in enumerate(sub_graphs):
    ax = fig.add_subplot(int(n_subgraphs / 3), 3, i)
```

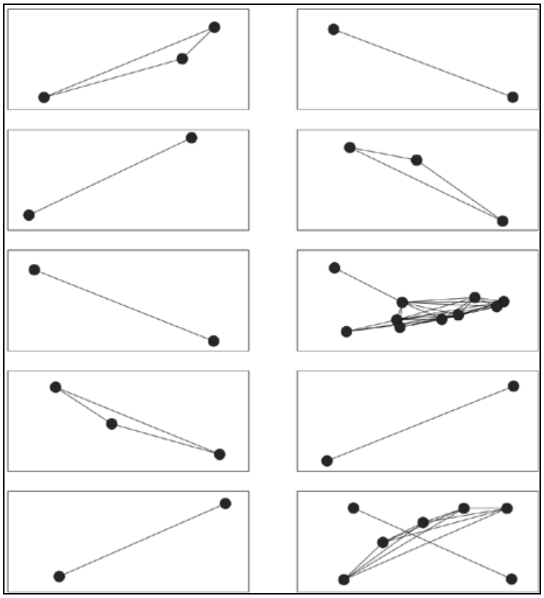
pyplot默认显示坐标轴标签，在这里没有实际意义。因此把这个功能关掉：

```
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```

然后再绘制顶点和边（用ax参数绘制相应的子图）。绘图之前需要设置好布局：

```
    pos = nx.spring_layout(G)
    nx.draw_networkx_nodes(G, pos, sub_graph.nodes(), ax=ax,
      node_size=500)
    nx.draw_networkx_edges(G, pos, sub_graph.edges(), ax=ax)
```

从图像中就能了解到各连通分支顶点数量及连接方式。



7.2.2 优化参数选取准则

连通分支查找算法依赖于阈值参数，由其决定是否在图中添加边。换句话说，阈值控制着找到的连通分支数量和连通分支的大小。既然阈值如此关键，阈值多大才是最合适的？这个问题主观性很强，没有明确的答案。它是任何聚类分析任务都要面对的一个主要问题。

然而，我们可以事先确定什么样的结果才是令人满意的，再根据理想中的结果来寻找选取参数的准则。作为一般性规则，它应该能够使得：

- 同一簇（连通分支）内的个体尽可能相似
- 不同簇内的个体尽可能不相似

轮廓系数（Silhouette Coefficient）就是一种对上述两点的量化方法。给定一个个体，其轮廓系数定义如下：

$$s = \frac{b-a}{\max(a,b)}$$

其中a为簇内距离，表示与簇内其他个体之间的平均距离。b为簇间距离，也就是与最近簇内各个个体之间的平均距离。

总轮廓系数为每个个体轮廓系数的均值。总轮廓系数接近最大值1时，表示每个簇内的个体相似度很高，不同簇之间距离较远。总轮廓系数接近0时，表示所有的簇重合在一起，各簇之间距离很小。总轮廓系数接近最小值–1时，表示个体出现在错误的簇内，把它们分到其他簇效果可能会更好。

利用这种选取准则，找到一种解决方案（找到合适的阈值），通过调整阈值，使总轮廓系数达到最大值。为此，创建一函数，接收阈值，计算总轮廓系数。

然后，把它传给SciPy的`optimize`模块的`minimize`函数，该函数通过调整参数值，找到函数的最小值。但我们想最大化总轮廓系数，可SciPy没有提供寻找最大值的函数。因此，对总轮廓系数取反后再找最小值（与直接求最大值是一回事）。

`scikit-learn`库提供计算轮廓系数的函数`sklearn.metrics.silhouette_score`，但是它与SciPy的`minimize`函数所要求的形式不一致。`minimize`函数要求变量参数（阈值）在其他参数的前面。这就需要把`friends`字典传进去计算图。函数声明如下：

```
def compute_silhouette(threshold, friends):
```

使用阈值参数创建图后，检查它是否至少有两个顶点：

```
    G = create_graph(friends, threshold=threshold)
    if len(G.nodes()) < 2:
```

轮廓系数的定义要求至少有两个顶点（才能计算距离）。如果少于两个顶点，认为这个问题无效，没法计算下去。有几种处理方法，最简单的是返回一个很小的值。轮廓系数最小值为–1，返回–99，表明问题无效。任何有效的值都比这个值大很多。代码如下：

```
        return -99
```

然后抽取连通分支：

```
sub_graphs = nx.connected_component_subgraphs(G)
```

轮廓系数的定义还要求至少有两个连通分支（才能计算不同簇之间的距离），并且至少其中一个连通分支有两个顶点（计算簇内距离）。检测这些条件是否满足，如果不满足，返回无效值。代码如下：

```
if not (2 <= nx.number_connected_components() < len(G.nodes())
  - 1):
    return -99
```

接着，需要获取标识着顶点被分到哪个连通分支的标签。遍历所有的连通分支，用字典保存顶点及其所属的连通分支。代码如下：

```
label_dict = {}
for i, sub_graph in enumerate(sub_graphs):
    for node in sub_graph.nodes():
        label_dict[node] = i
```

遍历图中所有顶点，依次获取到每个顶点的标签。需要分两步来做，先按一定顺序取到图，再遍历。因为图中顶点没有明确的顺序，但是只要没有改动图，顶点会维持现有顺序。这就表明，只要没有改动图，在图上调用`.nodes()`方法，返回的顶点顺序总是一致的。代码如下：

```
labels = np.array([label_dict[node] for node in G.nodes()])
```


注意，轮廓系数函数接收的是*距离矩阵*，而不是图。因此，要想办法把图转换为矩阵，同样分两步来做。首先，使用NetworkX的to_scipy_sparse_matrix函数把图转换为矩阵形式：

```
X = nx.to_scipy_sparse_matrix(G).todense()
```

写作本书时，scikit-learn实现的轮廓系数函数不支持稀疏矩阵。因此，需要调用todense()函数。很显然，这是个馊主意——通常应该用稀疏矩阵，因为数据密集的情况很罕见。这个例子这么用没问题，因为数据集相对较小；而再大点的数据集就不要这么做了。

> 建议你使用V-MEASURE或调整互信息（Adjusted Mutual Information）评价稀疏数据集。scikit-learn实现了这两种方法，但是两者所用到的参数差别很大。

请注意，图中边的权重表示相似度而不是距离。对于距离而言，值越大，相似度越小。可以用可能的最大相似度（1）减去现有相似度，把相似度转化为距离：

```
X = 1 - X
```

既然已经创建好距离矩阵和标签，现在就可以计算轮廓系数了。指定metric参数值为precomputed，否则X矩阵将被当作特征矩阵而不是距离矩阵（scikit-learn几乎到处都默认

使用的特征矩阵）。函数最后返回计算得到的轮廓系数：

```
    return silhouette_score(X, labels, metric='precomputed')
```

有两处用到取反操作。第一处是求距离时对相似度取反，这很有必要，因为scikit-learn计算轮廓系数的函数使用距离矩阵作为参数。第二处是对轮廓系数取反，这样才能用SciPy的optimize模块。

还有个小问题没有解决。上述函数返回的轮廓系数越大越好，而SciPy的optimize模块只定义了minimize函数，它是一个损失函数，值越小越好。需要对轮廓系数取反，定义一函数把打分函数转化为损失函数。

```
def inverted_silhouette(threshold, friends):
    return -compute_silhouette(threshold, friends)
```

上述函数为用原函数创建的一个新函数。新函数调用时，为原函数传入与之前相同的参数，只不过新函数在最后返回值时进行取反操作。

现在，就可以进行优化操作了。调用minimize函数，把用于取反操作的函数inverted_silhouette传进来：

```
result = minimize(inverted_silhouette, 0.1, args=(friends,))
```

参数说明如下。

- inverted_silhouette：对我们要最小化的函数compute_silhouette进行取反操作，将其变为损失函数。
- 0.1：我们一开始猜测阈值为0.1时，函数取到最小值。
- options={'maxiter':10}：只进行10轮迭代（增加迭代次数，效果可能更好，但运行时间也会相应增加）。
- method='nelder-mead'：使用下山单纯形法（Nelder-Mead）优化方法（SciPy提供的优化方法）。
- args=(friends,)：向被优化的函数传入friends字典参数。

上述程序运行时间较长。创建图的函数不是很快，计算轮廓系数的函数也不快。减少maxiter的值，减少迭代次数，能缩短运行时间，但那样很可能找到的是次优阈值。

运行完上述函数，我得到的最优阈值为0.135，这时有10个连通分支。最小化函数返回的值是–0.192，然而不要忘记前面进行了取反操作，所以实际的轮廓系数是0.192。这个值为正数，表明各簇比起重合更像是分散的（好事）。可以运行其他模型，看下轮廓系数是否更大，各簇是否

更分散。

可以把这个结果用到用户推荐上——如果用户属于一个连通分支,可以向他推荐该连通分支的其他用户。简要回顾下针对用户推荐的研究历程,先是用杰卡德相似系数寻找用户间的相似度,再用连通分支把用户分成不同的簇，然后使用最优化方法找到最好的模型。

然而,大量用户可能与其他用户没有很好的联系,因此需要用不同的算法找到他们所属的簇。

7.3 小结

本章探讨了社交网络图以及如何对其进行聚类分析。还探讨了如何加载和使用在第6章创建的分类模型。

用来自Twitter的数据，创建了好友关系图，根据用户的好友，检测用户之间的相似度。我们认为共同好友更多的用户更相似，考虑到好友数量可能差别很大，对其进行了规范化处理。根据用户的相似度进行推测，是一种常用的知识（比如年龄或谈论的主题）推断方法。本章用下面的逻辑向用户推荐好友——如果他们关注用户X，用户Y和X相似，那么他们可能喜欢用户Y。这种方法与前几章从交易数据中挖掘相似商品有异曲同工之妙。

我们的目标是推荐用户，使用聚类分析方法能够找到不同的用户簇，主要步骤有根据相似度创建加权图，从图中寻找连通分支。创建图时用到了`NetworkX`库。

用轮廓系数评价聚类效果，其原则是簇内距离最小和簇间距离最大。轮廓系数越大，聚类效果越好。在寻找最合适的阈值时，用到了SciPy的`optimize`模块。

本章还比较了几对意义相反的概念。对于两者之间的*相似度*这个概念，值越大，表明两者之间更相像。相反，对于*距离*而言，值越小，两者更相像。另外一对是*损失函数*和*打分函数*。对于*损失函数*，值越小，效果越好（也就是损失越少）。而对于*打分函数*，值越大，效果越好。

下一章将介绍怎样从另外一种数据类型——图像中抽取特征。接下来，将讨论如何用神经网络识别图像中的数字，编写程序实现自动化破解验证码。

第 8 章

用神经网络破解验证码

理解图像中的信息一直是数据挖掘领域的一个难题，直到最近几年才开始得到真正解决。图像检测和理解算法已相当成熟，几大厂商使用这些算法研制的监测系统已投入商用，用来处理实际问题。这些系统能够理解和识别视频画面中的人和物体。

从图像中抽取信息很难。图像包含大量原始数据，图像的标准编码单元——像素——提供的信息量很少。图像——特别是照片——可能存在一系列问题，比如模糊不清、离目标太近、光线很暗或太亮、比例失真、残缺、扭曲等，这会增加计算机系统抽取有用信息的难度。

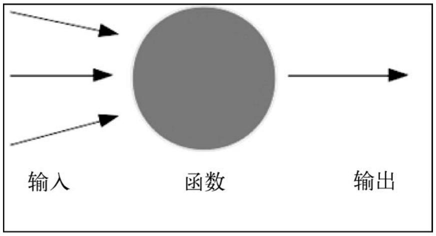
本章介绍如何用神经网络识别图像中的字母，从而自动识别验证码。验证码的设计初衷是便于人类理解，而不易被计算机识破。验证码的英文名叫作CAPTCHA，它取自以下短语中几个单词的首字母“Completely Automated Public Turing test to tell Computers and Humans Apart”，意思是能够区别计算机和人类的全自动的公共图灵测试。很多网站都在注册、评论系统中使用验证码，以防止别人恶意注册虚假账号或发布垃圾评论。

本章主要介绍如下内容。

- 神经网络
- 创建验证码和字母数据集
- 用`scikit-image`库处理图像数据
- 神经网络库`PyBrain`
- 从图像中抽取基本特征
- 使用神经网络进行更大规模的分类任务
- 用后处理提升效果

8.1 人工神经网络

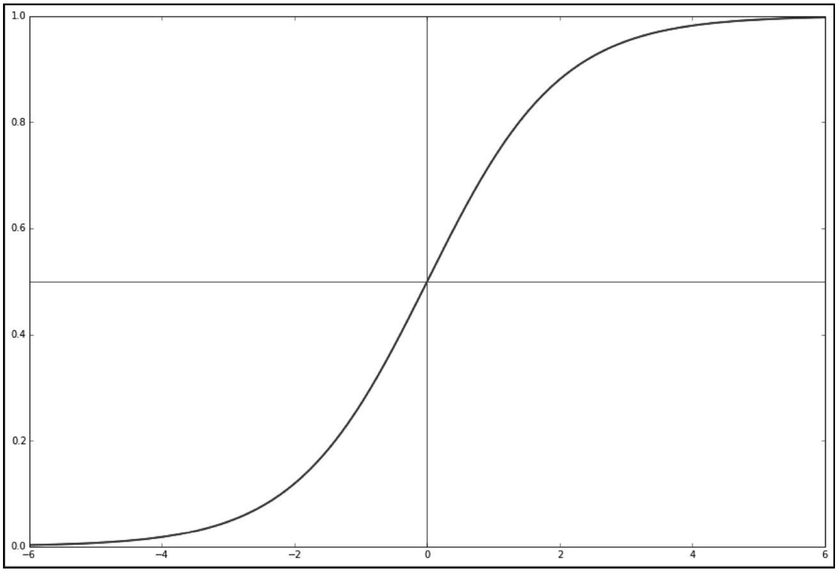
神经网络算法最初是根据人类大脑的工作机制设计的。然而，该领域近年所取得的进展主要得益于数学而不是生物学。神经网络由一系列相互连接的神经元组成。每个神经元都是一个简单的函数，接收一定输入，给出相应输出。



神经元可以使用任何标准函数来处理数据，比如线性函数，这些函数统称为*激活函数*（activation function）。一般来说，神经网络学习算法要能正常工作，激活函数应当是可导（derivable）和光滑的。常用的激活函数有逻辑斯谛函数，函数表达式如下（x为神经元的输入，k、L通常为1，这时函数达到最大值）。

$$f(x)=\frac{L}{1+e^{-k(x-x_0)}}$$

下图为x取−6到6之间的函数图像。



两条红线的交点表示x为0时，函数值为0.5。

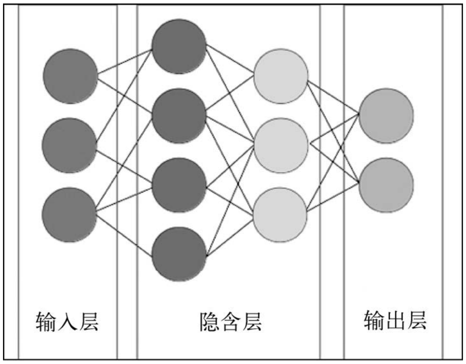
每个神经元接收几个输入，根据这几个输入[①]，计算输出。这样的一个个神经元连接在一起组成了神经网络，对数据挖掘应用来说，它非常强大。这些神经元紧密连接，密切配合，能够通过学习得到一个模型，使得神经网络成为机器学习领域最强大的概念之一。

① 对带权重的输入加总。——译者注

神经网络简介

用于数据挖掘应用的神经网络，神经元按照层级进行排列。第一层，也就是*输入层*，接收来自数据集的输入。第一层中的每个神经元对输入进行计算，把得到的结果传给第二层的神经元。这种叫作*前向神经网络*。本章暂且把它简称为神经网络。此外，还有几种神经网络，适用于不同的应用。第11章会讲到另外一种类型的神经网络。

神经网络中，上一层的输出作为下一层的输入，直到到达最后一层：*输出层*。输出结果表示的是神经网络分类器给出的分类结果。输入层和输出层之间的所有层被称为*隐含层*，因为在这些层中，其数据表现方式，常人难以理解。大多数神经网络至少有三层，而如今大多数应用所使用的神经网络层次比这多得多。



我们优先考虑使用全连接层，即上一层中每个神经元的输出都输入到下一层的所有神经元。实际构建神经网络时，就会发现，训练过程中，很多权重都会被设置为0，有效地减少边的数量。比起其他连接模式，全连接神经网络更简单，计算起来更快捷。

神经元激活函数通常使用逻辑斯谛函数，每层神经元之间为全连接，创建和训练神经网络还需要用到其他几个参数。创建过程，指定神经网络的规模需要用到两个参数：神经网络共有多少层，隐含层每层有多少个神经元（输入层和输出层神经元数量通常由数据集来定）。

训练过程还会用到一个参数：神经元之间边的权重。一个神经元连接到另外一个神经元，两者之间的边具有一定的*权重*，在计算输出时，用边的权重乘以信号的大小（signal，第一个神经元的输出）。如果边的权重为0.8，神经元激活后，输出值为1，那么下一个神经元从前面这个神经元得到的输入就是0.8。如果第一个神经元没有激活，值为0，那么输出到第二个神经元的值就是0。

神经网络大小合适，且权重经过充分训练，它的分类效果才能精确。大小合适并不是越大越好，因为神经网络过大，训练时间会很长，更容易出现过拟合训练集的情况。

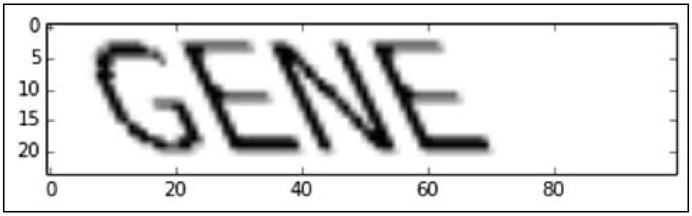
> 通常，开始时使用随机选取的权重，训练过程中再逐步更新。

设置好第一个参数（网络的大小），再从训练集中训练得到边的权重参数后，就能构造分类器。然后，就可以用它进行分类，跟前几章用到的分类器算法很像。但是，首先需要准备训练集和测试集。

8.2 创建数据集

本章，我们将扮演骇客这个角色，我们想编写破解网站验证码的程序，这样我们就有能力在别人网站上发布恶意广告。需要注意的是，我们要破解的验证码，难度比不上现在网上常用的；还有就是发表垃圾广告不太道德。

我们只使用长度为4个字母的英文单词作为验证码，如下图所示。



我们的目标是编写程序还原图像中的单词，步骤如下。

(1) 把大图像分成只包含一个字母的4张小图像。
(2) 为每个字母分类。
(3) 把字母重新组合为单词。
(4) 用词典修正单词识别错误。

我们的验证码破解算法做出了以下几个假设。首先，验证码中的单词是一个完整的、有效的英文单词，其长度为4个字母（实际上，生成和破解验证码，我们都使用同一个词典）。其次，单词全部字母均为大写形式，不使用符号、数字或空格。为了增加难度，在生成图像时对单词使用不同的错切（shear）变化效果。

8.2.1 绘制验证码

接下来，我们编写创建验证码的函数，目标是绘制一张含有单词的图像，对单词使用错切变化效果。绘制图像要用到`PIL`库，错切变化需要使用`scikit-image`库。`scikit-image`库能够接收`PIL`库导出的`numpy`数组格式的图像数据，因此这两个工具可以结合使用。

> PIL和scikit-image库都可以用pip安装：
> ```
> pip install PIL
> pip install scikit-image
> ```

首先，导入即将用到的库和模块。导入numpy，从PIL中导入Image等绘图函数。

```
import numpy as np
from PIL import Image, ImageDraw, ImageFont
from skimage import transform as tf
```

接着，创建用于生成验证码的基础函数。这个函数接收一个单词和错切值（通常在0到0.5之间），返回用numpy数组格式表示的图像。该函数还提供指定图像大小的参数，因为后面还会用它生成只包含单个字母的测试数据。代码如下：

```
def create_captcha(text, shear=0, size=(100, 24)):
```

我们使用字母L来生成一张黑白图像，为ImageDraw类初始化一个实例。这样，我们就可以用PIL绘图。代码如下：

```
    im = Image.new("L", size, "black")
    draw = ImageDraw.Draw(im)
```

指定验证码文字所使用的字体。这里要用到字体文件，下面代码中的文件名（Coval.otf）应该指向文件存放位置（我把它放到当前笔记本所在目录）。

```
    font = ImageFont.truetype(r"Coval.otf", 22)
    draw.text((2, 2), text, fill=1, font=font)
```

> 你可以从开源字库（Open Font Library）下载所需字体文件Coval.otf：http://openfontlibrary.org/en/font/bretan。

把PIL图像转换为numpy数组，以便用scikit-image库为图像添加错切变化效果。scikit-image大部分计算都使用numpy数组格式。代码如下：

```
    image = np.array(im)
```

然后，应用错切变化效果，返回图像。

```
    affine_tf = tf.AffineTransform(shear=shear)
    image = tf.warp(image, affine_tf)
    return image / image.max()
```

上面最后一行代码对图像特征进行归一化处理，确保特征值落在0到1之间。归一化处理可在数据预处理、分类或其他阶段进行。

现在，我们可以轻松生成图像，使用pyplot绘制图像。首先设定魔术方法%matplotlib，

指定在当前笔记本作图，导入pyplot。代码如下：

```
%matplotlib inline
from matplotlib import pyplot as plt
```

生成验证码图像并显示它。

```
image = create_captcha("GENE", shear=0.5)
plt.imshow(image, cmap='Greys')
```

生成的图像就是本节开头你见到的那张，效果还不错吧。

8.2.2 将图像切分为单个的字母

虽然我们的验证码是单词，但是我们不打算构造能够识别成千上万个单词的分类器，而是把大问题转化为更小的问题：识别字母。

验证码识别算法的下一步是分割单词，找到其中的字母。具体做法是，创建一个函数，寻找图像中连续的黑色像素，抽取它们作为新的小图像。这些小图像（或者至少应该）就是我们要找的字母。

首先，导入图像分割函数要用到的label、regionprops函数。

```
from skimage.measure import label, regionprops
```

图像分割函数接收图像，返回小图像列表，每张小图像为单词的一个字母，函数声明如下：

```
def segment_image(image):
```

我们要做的第一件事就是检测每个字母的位置，这就要用到scikit-image的label函数，它能找出图像中像素值相同且又连接在一起的像素块。这有点像第7章中的连通分支。

label函数的参数为图像数组，返回跟输入同型的数组。在返回的数组中，图像连接在一起的区域用不同的值来表示，在这些区域以外的像素用0来表示。代码如下：

```
    labeled_image = label(image > 0)
```

抽取每一张小图像，将它们保存到一个列表中。

```
    subimages = []
```

scikit-image库还提供抽取连续区域的函数：regionprops。遍历这些区域，分别对它们进行处理。

```
    for region in regionprops(labeled_image):
```

这样，通过region对象就能获取到当前区域的相关信息。我们这里要用到当前区域的起始和结束位置的坐标。

```
        start_x, start_y, end_x, end_y = region.bbox
```

用这两组坐标作为索引就能抽取到小图像（image对象为numpy数组，可以直接用索引值），然后，把它保存到subimages列表中。代码如下：

```
        subimages.append(image[start_x:end_x,start_y:end_y])
```

最后（循环外面），返回找到的小图像，每张（希望如此）小图像包含单词的一个字母区域。没有找到小图像的情况，直接把原图像作为子图返回。代码如下：

```
    if len(subimages) == 0:
        return [image,]
    return subimages
```

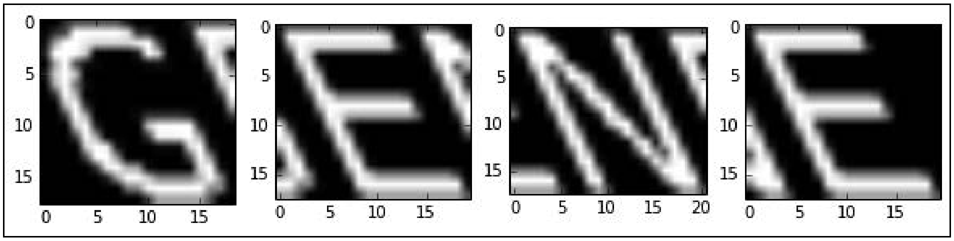
使用刚定义的这个函数，就能从前面生成的验证码中找到小图像。

```
subimages = segment_image(image)
```

还可以像下面这样查看每张小图像。

```
f, axes = plt.subplots(1, len(subimages), figsize=(10, 3))
for i in range(len(subimages)):
    axes[i].imshow(subimages[i], cmap="gray")
```

结果如下。



图像切割效果还不错，但是你可能注意到，每张小图像都多少带有相邻字母的一部分。

8.2.3　创建训练集

使用图像切割函数就能创建字母数据集，其中字母使用了不同的错切效果。然后，就可以训练神经网络分类器来识别图像中的字母。

首先，指定随机状态值，创建字母列表，指定错切值。这里几乎没有新内容，numpy的arange函数你可能没用过，它跟Python的range函数类似——只不过arange函数可以和numpy的数组一起用，步长可以使用浮点数。代码如下：

```
from sklearn.utils import check_random_state
random_state = check_random_state(14)
```

```
letters = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
shear_values = np.arange(0, 0.5, 0.05)
```

再来创建一个函数（用来生成一条训练数据），从我们提供的选项中随机选取字母和错切值。代码如下：

```
def generate_sample(random_state=None):
    random_state = check_random_state(random_state)
    letter = random_state.choice(letters)
    shear = random_state.choice(shear_values)
```

返回字母图像及表示图像中字母属于哪个类别的数值。字母A为类别0，B为类别1，C为类别2，以此类推。代码如下：

```
    return create_captcha(letter, shear=shear, size=(20, 20)),
      letters.index(letter)
```

在上述函数体的外面，调用该函数，生成一条训练数据，用pyplot显示图像。

```
image, target = generate_sample(random_state)
plt.imshow(image, cmap="Greys")
print("The target for this image is: {0}".format(target))
```

调用几千次该函数，就能生成足够的训练数据。把这些数据传入到numpy的数组里，因为数组操作起来比列表更容易。代码如下：

```
dataset, targets = zip(*(generate_sample(random_state) for i in
range(3000)))
dataset = np.array(dataset, dtype='float')
targets =  np.array(targets)
```

我们共有26个类别，每个类别（字母）用从0到25之间的一个整数表示。神经网络一般不支持一个神经元输出多个值，但是多个神经元就能有多个输出，每个输出值在0到1之间。因此，我们对类别使用一位有效码编码方法，这样，每条数据就能得到26个输出。如果结果像某字母，使用近似于1的值；如果不像，就用近似于0的值。代码如下：


```
from sklearn.preprocessing import OneHotEncoder
onehot = OneHotEncoder()
y = onehot.fit_transform(targets.reshape(targets.shape[0],1))
```

我们用的库不支持稀疏矩阵，因此，需要将稀疏矩阵转换为密集矩阵。代码如下：

```
y = y.todense()
```

8.2.4 根据抽取方法调整训练数据集

得到的数据集跟即将使用的方法有较大出入。数据集每条数据都是一个恰好为20像素见方的字母。我们所使用的方法是从单词中抽取字母，而这可能会挤压图像，使图像偏离中心或者引入

其他问题。

理想情况下，训练分类器所使用的数据应该与分类器即将处理的数据尽可能相似。但实际中，经常有所不同，但是应尽量缩小两者之间的差别。

对于验证码识别实验，最理想的情况是，从实际的验证码中抽取字母，并对它们进行标注。为了节省时间，我们只在训练集上运行分割函数，返回分割后得到的字母图像 。

我们需要用到scikit-image库中的resize函数，因为我们得到的小图像并不总是20像素见方。代码如下：

```
from skimage.transform import resize
```

现在，就可以对每条数据运行segment_image函数，将得到的小图像调整为20像素见方。代码如下：

```
dataset = np.array([resize(segment_image(sample)[0], (20, 20)) for
sample in dataset])
```

最后，创建我们的数据集。dataset数组为三维的，因为它里面存储的是二维图像信息。由于分类器接收的是二维数组，因此，需要将最后两维扁平化。

```
X = dataset.reshape((dataset.shape[0], dataset.shape[1] * dataset.
shape[2]))
```

使用scikit-learn中的train_test_split函数，把数据集切分为训练集和测试集。代码如下：

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, train_size=0.9)
```

8.3 训练和分类

接下来，我们就来构造神经网络分类器，接收图像，预测图像中的（单个）字母是什么。

我们将使用之前创建的单个字母训练集。数据集本身很简单。每张图像为20像素大小，每个像素用1（黑）或0（白）来表示。每张图像有400个特征，将把它们作为神经网络的输入。输出结果为26个0到1之间的值。值越大，表示图像中的字母为该值所对应的字母（输出的第一个值对应字母A，第二个对应字母B，以此类推）的可能性越大。

我们用PyBrain库来构建神经网络分类器。

> 跟我们之前见到的所有库一样，PyBrain也可以用pip来安装：pip install pybrain。

PyBrain库使用自己的数据集格式，好在创建这种格式的训练集和测试集并不太难。代码如下：

```
from pybrain.datasets import SupervisedDataSet
```

首先，遍历我们的训练集，把每条数据添加到一个新的SupervisedDataSet实例中。代码如下：

```
training = SupervisedDataSet(X.shape[1], y.shape[1])
for i in range(X_train.shape[0]):
    training.addSample(X_train[i], y_train[i])
```

然后，遍历测试集，同样把每条数据添加到一个新的SupervisedDataSet实例中。代码如下：

```
testing = SupervisedDataSet(X.shape[1], y.shape[1])
for i in range(X_test.shape[0]):
    testing.addSample(X_test[i], y_test[i])
```

现在就可以创建神经网络了。我们将创建一个最基础的、具有三层结构的神经网络，它由输入层、输出层和一层隐含层组成。输入层和输出层的神经元数量是固定的。数据集有400个特征，那么第一层就需要有400个神经元，而26个可能的类别表明我们需要26个用于输出的神经元。

确定隐含层的神经元数量可能相当困难。如果神经元数量过多，神经网络呈现出稀疏的特点，这时要训练足够多的神经元恰当地表示数据就有困难，这往往会导致过拟合训练数据的问题。相反，如果神经元过少，每个对分类结果的贡献过大，再加上训练不充分，就很可能产生低拟合现象。我发现一开始用漏斗形状不错，即隐含层神经元数量介于输入和输出层之间。本章，隐含层用100个神经元，你可以尝试其他值，看看能不能取得更好的效果。

导入buildNetwork函数，指定维度，创建神经网络。第一个参数X.shape[1]为输入层神经元的数量，也就是特征数（数据集X的列数）。第二个参数指隐含层的神经元数量，这里设置为100。第三个参数为输出层神经元数量，由类别数组y的形状来确定。最后，除去输出层外，我们每层使用一个一直处于激活状态的偏置神经元（bias neuron，它与下一层神经元之间有边连接，边的权重经过训练得到）。代码如下：

```
from pybrain.tools.shortcuts import buildNetwork
net = buildNetwork(X.shape[1], 100, y.shape[1], bias=True)
```

现在，我们就可以训练神经网络，寻找合适的权重值。但是如何训练神经网络呢？

8.3.1　反向传播算法

反向传播算法（back propagation，backprop）的工作机制为对预测错误的神经元施以惩罚。从输出层开始，向上层层查找预测错误的神经元，微调这些神经元输入值的权重，以达到修复输出错误的目的。

神经元之所以给出错误的预测，原因在于它前面为其提供输入的神经元，更确切来说是由这两个神经元之间边的权重及输入值决定的。我们可以尝试对权重进行微调。每次调整的幅度取决于以下两个方面：神经元各边权重的误差函数的偏导数和一个叫作*学习速率*的参数（通常使用很小的值）。计算出函数误差的梯度，再乘以学习速率，用总权重减去得到的值。下面将给出例子。梯度的符号由误差决定，每次对权重的修正都是*朝着*给出正确的预测值努力。有时候，修正结果为*局部最优*（local optima），比起其他权重组合要好，但所得到的各权重还不是最优组合。

反向传播算法从输出层开始，层层向上回溯到输入层。到达输入层后，所有边的权重更新完毕。

`PyBrain`提供了backprop算法的一种实现，在神经网络上调用`trainer`类即可。代码如下：

```
from pybrain.supervised.trainers import BackpropTrainer
trainer = BackpropTrainer(net, training, learningrate=0.01,
weightdecay=0.01)
```

backprop算法在训练集上进行迭代，每次都对权重进行微调。当误差减小的幅度很小时，表明算法无法进一步缩小误差，继续训练已无意义，这时就可以停止算法运行。理论上，等误差不再缩小时，再停止运行。这个过程叫作算法收敛。但实际上我们不会等到误差停止缩小，因为这通常要花很长时间且效果有限。

此外，更简单的做法是，运行代码固定的*步数*（epoch）。每一步所包含的次数越多，所用时间也就越多，效果越好（每一步的提升效果呈下降趋势）。我们这里训练时，使用了20步，数量再多些，效果可能会有所改善（也许幅度很小）。代码如下：

```
trainer.trainEpochs(epochs=20)
```

运行前面的代码，这可能要花几分钟时间，长短取决于硬件，然后我们就能在测试集上进行预测。`PyBrain`提供了相应函数，只要在`trainer`实例上调用即可。

```
predictions = trainer.testOnClassData(dataset=testing)
```

得到预测值后，可以用`scikit-learn`计算F1值。

```
from sklearn.metrics import f1_score
print("F-score: {0:.2f}".format(f1_score(predictions,
                                          y_test.argmax(axis=1) )))
```

F1值为0.97，对于相对较小的模型来说，这个结果很了不起。别忘了我们的特征值只是简单

的像素值，神经网络竟然知道怎么使用它们。

既然构建好了具有高精度的字母分类器，接下来看下怎么用它来识别单词，实现验证码破解功能。

8.3.2 预测单词

我们想分别识别每张小图像中的字母，然后把它们拼成单词，完成验证码识别。

我们来定义一个函数，接收验证码，用神经网络进行训练，返回单词预测结果。

```
def predict_captcha(captcha_image, neural_network):
```

使用前面定义的图像切割函数segment_image抽取小图像。

```
    subimages = segment_image(captcha_image)
```

最终的预测结果——单词，将由小图像中的字母组成。这些小图像根据位置进行排序，从而保证拼接后得到的单词中各字母处在正确的位置上。

```
    predicted_word = ""
```

遍历四张小图像。

```
    for subimage in subimages:
```

每张小图像不太可能正好是20像素见方。调整大小后，才能用神经网络处理。

```
        subimage = resize(subimage, (20, 20))
```

把小图像数据传入神经网络的输入层，激活神经网络。这些数据将在神经网络中进行传播，返回输出结果。神经网络在前面已经创建好了，现在只需激活它。代码如下：


```
        outputs = net.activate(subimage.flatten())
```

神经网络输出26个值，每个值都有索引号，分别对应letters列表中有着相同索引的字母，每个值的大小表示与对应字母的相似度。为了获得实际的预测值，我们取到最大值的索引，再通过letters列表找到对应的字母。例如，如果第五个值最大，那么预测结果就为字母E。代码如下：

```
        prediction = np.argmax(outputs)
```

把上面得到的字母添加到正在预测的单词中。

```
        predicted_word += letters[prediction]
```

循环结束后，我们就已经找到了单词的各个字母，将其拼接成单词，最后返回这个单词。

```
    return predicted_word
```

可以使用下面代码来做下测试。尝试不同的单词，看看可能会遇到什么错误，别忘了我们的神经网络只能处理大写字母。

```
word = "GENE"
captcha = create_captcha(word, shear=0.2)
print(predict_captcha(captcha, net))
```

我们可以把上面代码整合到一个函数里，便于进行多次尝试。这里沿用之前每个单词只包含四个字母的假设，降低预测任务的难度。删除`prediction = prediction[:4]`，试试看可能会出什么错误。代码如下：

```
def test_prediction(word, net, shear=0.2):
    captcha = create_captcha(word, shear=shear)
    prediction = predict_captcha(captcha, net)
    prediction = prediction[:4]
    return word == prediction, word, prediction
```

上述函数返回结果依次为预测结果是否正确、验证码中的单词和预测结果的前四个字符。

上面代码能正确识别单词GENE，但是其他单词会出错。正确率如何？我们借助`NLTK`模块创建单词数据集，只使用长度为4的单词。从`NLTK`中导入`words`，代码如下：

```
from nltk.corpus import words
```

`words`实际上为`corpus`（语料库）对象，调用它的`words()`方法才能取到里面的单词。下面代码还加了长度为4的过滤条件。代码如下：

```
valid_words = [word.upper() for word in words.words() if len(word) ==
4]
```

识别得到的所有长度为4的单词，分别统计识别正确和错误的数量。

```
num_correct = 0
num_incorrect = 0
for word in valid_words:
    correct, word, prediction = test_prediction(word, net,
                                                shear=0.2)
if correct:
        num_correct += 1
    else:
        num_incorrect += 1
print("Number correct is {0}".format(num_correct))
print("Number incorrect is {0}".format(num_incorrect))
```

正确识别2832个单词，错误识别2681个，正确率仅为51%。而字母识别率为97%，两者差距太大。到底怎么回事？

首先，来看下正确率本身。其余条件相同的情况下，我们有四个字母，每个字母的正确率为

97%，四个字母都正确的话，正确率约为88%（约为0.97^4）。一个字母出错将导致整个单词识别错误。

其次，错切值对正确率有影响。这次创建数据集时，随机从0到0.5之间选取一个数作为错切值。先前测试时错切值为0.2。错切值为0时，正确率为75%；错切值取0.5时，正确率只有2.5%。错切值越大，正确率越低。

另外一个原因在于我们之前随机选取字母组成单词，而字母在单词中的分布不是随机的。例如，字母E显然就比Q等其他字母使用频率更高。使用频度较高，但却常常被识别错误的字母，也会导致错误率上升。

我们可以把经常识别错误的字母统计出来，用二维混淆矩阵来表示。每行和每列均为一个类别（字母）。

矩阵的每一项表示一个类别（行对应的类）被错误识别为另一个类别（列对应的类）的次数。例如，`(4,2)`这一项的值为6，表示字母D有6次被错误地识别为字母B。

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(np.argmax(y_test, axis=1), predictions)
```

理想情况下，混淆矩阵应该只有处于对角线的项`(i,i)`值不为零，其余各项的值均为零，否则就是分类有误。

用`pyplot`做成图，哪些字母容易混淆就一目了然，代码如下：

```
plt.figure(figsize=(10, 10))
plt.imshow(cm)
```

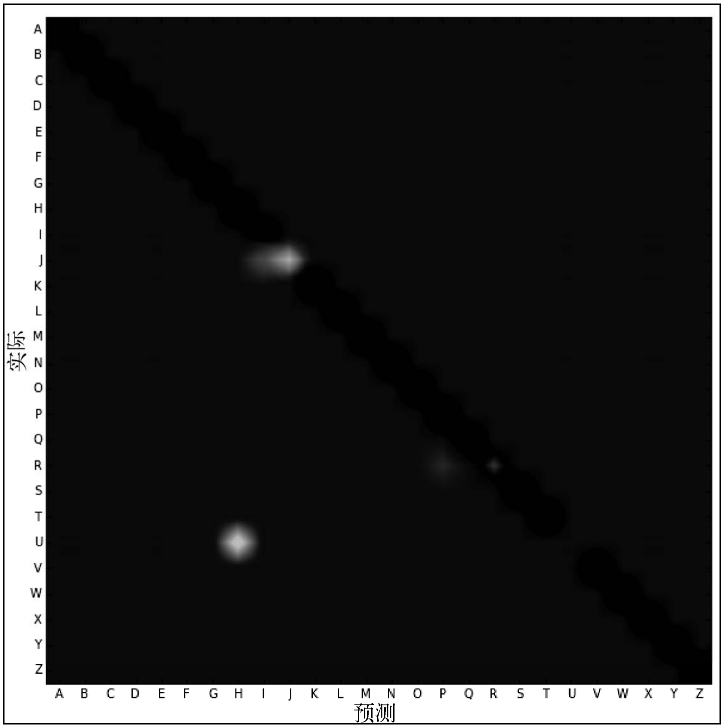
下面代码中的`tick_marks`表示刻度，在坐标轴上标出每个字母，便于理解。

```
tick_marks = np.arange(len(letters))
plt.xticks(tick_marks, letters)
plt.yticks(tick_marks, letters)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

结果如下图所示。从图中，我们就能清楚地看到主要的错误在于几乎无一例外地把U识别为H！

我们的词表中17%的单词含有字母U，这些单词几乎都会被识别错误。U的出现频率要高于H（11%的单词），我们不禁想到了一个提升正确率的简单方法（可能不够健壮）：把所有预测结果为H的，都改为U。

下节将采用更为巧妙的方法，从词典中搜索跟预测结果相似的单词。



8.4　用词典提升正确率

我们刚刚是直接返回预测结果，其实返回之前可以先检查一下词典里是否包含该词条。如果单词在词典里，那么就返回预测结果，如果不在，我们找到和预测结果相似的单词，再把它作为更新过的预测结果返回。注意，该方法是基于所有验证码中的单词都是英语单词的假设，因此，对于随机选取字母组成的验证码来说不适用。其实这正是很多验证码不使用单词的原因之一。

还有一个问题要解决——如何找到最相近的单词？方法有很多。例如，比较词长。词长差不多的两个单词更相似。然而，通常认为，在相同位置有相同字母的两个单词更相似，这就要用到编辑距离。

8.4.1　寻找最相似的单词

列文斯坦编辑距离（Levenshtein edit distance）是一种通过比较两个短字符串，确定它们相似度的方法。它不太适合扩展，字符串很长时通常不用这种方法。编辑距离需要计算从一个单词变为另一个单词所需要的步骤数。以下操作都算一步。

(1) 在单词的任意位置插入一个新字母。

(2) 从单词中删除任意一个字母。
(3) 把一个字母替换为另外一个字母。

把一个单词转变为另外一个单词所需步数越少，这两个单词越相似，反之，差别越大。

NLTK实现了编辑距离算法，可以直接拿来用。导入后，指定两个单词，看下它们的编辑距离。

```
from nltk.metrics import edit_distance
steps = edit_distance("STEP", "STOP")
print("The number of steps needed is: {0}".format(steps))
```

尝试不同的单词，你会发现编辑距离返回的结果跟我们的直觉很相近。编辑距离在检查拼写、听写错误和名称匹配（比如Marc和Mark的拼写形式就极易被搞混）方面用处很大。

然而，我们这里没这么复杂，只需要比较两个单词同等位置上的字母是否一致即可，不涉及字母的移动。因此，我们自己来编写距离度量函数，也就是统计同等位置上字母不同的数量。代码如下：

```
def compute_distance(prediction, word):
    return len(prediction) - sum(prediction[i] == word[i] for i in
range(len(prediction)))
```

用词长（4）减去同等位置上相同的字母的数量，得到的值越小表示两个词相似度越高。

8.4.2　组装起来

我们现在就来测试改进的预测函数，代码跟之前相似。首先，定义预测函数，这次把单词列表也传进去。

```
from operator import itemgetter
def improved_prediction(word, net, dictionary, shear=0.2):
    captcha = create_captcha(word, shear=shear)
    prediction = predict_captcha(captcha, net)
    prediction = prediction[:4]
```

到目前为止，代码跟之前的预测函数一致，还是取到预测结果的前四个字母。然而，这次还要检测单词是否在词典里。如果在，把单词作为预测结果返回；如果不在，找到最相似的单词返回。代码如下：

```
    if prediction not in dictionary:
```

计算预测结果和词典中每个单词的距离，按照距离由小到大进行排序。代码如下：

```
        distances = sorted([(word, compute_distance(prediction, word))
                             for word in dictionary],
key=itemgetter(1))
```

找到最为匹配的单词——也就是距离最小的单词——随后把这个单词作为预测结果返回。

```
        best_word = distances[0]
        prediction = best_word[0]
```

函数返回结果跟之前相同：预测结果是否正确，验证码中的单词和预测结果。

```
    return word == prediction, word, prediction
```

测试代码中需要变动的地方用粗体表示。

```
num_correct = 0
num_incorrect = 0
for word in valid_words:
    correct, word, prediction = improved_prediction(word, net, valid_
words, shear=0.2)
    if correct:
        num_correct += 1
    else:
        num_incorrect += 1
print("Number correct is {0}".format(num_correct))
print("Number incorrect is {0}".format(num_incorrect))
```

上述代码运行时间稍长（计算预测结果跟词典中每个单词之间的距离要耗费一定时间），最终正确识别3037个单词，错误识别2476个，正确率较之前有4个百分点的提升，达到55%。由于预测结果跟词典中很多单词的相似度一致，算法随机从一组最相似的单词中选择最佳的，所以算法效果提升很有限。例如，词典中第一个单词AANI（埃及神话中狗头猿猴），跟其他44个单词距离相同。选中的几率就只有1/44。

如果我们作弊，预测结果只要从最相似的一组词中随机挑一个就算是正确的话，那么预测结果正确率将高达78%（代码请见本书配套的代码包）。

要进一步提升正确率，得改变距离测量方法，看能不能利用之前得到的混淆矩阵，找到经常被混淆的字母或其他有用信息。逐步改进效果，是很多数据挖掘算法的一个特点，这跟科学实验方法是一脉相承的——产生想法，做实验，分析结果，再根据结果做进一步的改进。

8.5　小结

本章的任务是根据验证码图像的像素值识别里面的单词。当然，为了简化难度，验证码使用由四个字母组成的英语单词。网站上实际使用的验证码要比这复杂得多——也应该这样！但是，只要对上面所讲的算法进行改进，就能通过神经网络，使用跟这里类似的方法破解更为复杂的验证码。`scikit-image`库提供大量图像处理工具，比如从图像中抽取不同形状的小图像和改善图像对比度的方法等。这些方法有助于破解验证码。

我们把识别单词的难题转化为识别字母的小问题，创建前向神经网络识别图像中的字母，取

得了97%的准确率。

神经网络由一组组神经元连接而成，神经元为基本的计算单元，每个神经元都只包含一个函数。然而，所有的神经单元连接在一起就能解决复杂程度很高的问题。神经网络是深度学习的基础，后者可是当今数据挖掘领域最受关注的方向之一。

尽管识别字母正确率很高，但是单词识别正确率陡然降至50%多点。原因有很多，其实这恰好说明把算法从实验环境搬到真实环境会遇到各种各样的困难。

使用词典，寻找最匹配的单词，能够提升正确率。我们考虑使用常用的编辑距离表示单词间的相似度；但是我们只关注字母错误而不深究与哪些编辑步骤（插入、删除）相关，因此简化了距离计算方法。最终正确率有所提升，但是还有很多要改进的地方。

下一章将继续研究字符串比较，尝试找出文档的主人（从多名可能的作者中）——只根据文档内容而不使用其他信息！

第 9 章

作者归属问题

文本挖掘任务作者分析（authorship analysis）的目标是只根据作品内容找出作者独有的特点，比如年龄、性别或背景。*作者归属*（authorship attribution）是作者分析的一个细分领域，研究目标是从一组可能的作者中找到文档真正的主人。这是一种典型的分类任务。作者分析任务一般采用标准的数据挖掘方法，比如交叉检验、特征抽取和分类算法等。

本章将把前几章学到的数据挖掘方法整合起来，解决作者归属问题，从而掌握数据挖掘整个过程。我们界定问题，讨论相关背景和知识，抽取特征，创建流水线实现分类。我们将比较两类特征的效果：功能词和N元语法。最后，深入分析结果。数据集会用到两种，一开始使用图书，然后增加难度，使用噪音较多的真实电子邮件语料。

本章主要介绍如下内容。

- 特征工程和如何根据应用选择特征
- 带着新问题，重新回顾词袋模型
- 特征类型和字符N元语法模型
- 支持向量机
- 数据集清洗

9.1 为作品找作者

作者分析可以从*文体学*（stylometry）中找到它的一席之地，文体学分析研究的是作者的写作风格，所依据的理念是每个人在语言掌握上有微小差异，这会反映到写作中，通过分析作品之间的细微差别，就可以把作者区分开来。

过去一直靠人工统计、分析来解决作者分析问题，而这些工作恰好是计算机所擅长的，这表明可以借助数据挖掘技术来实现自动化。如今作者分析的研究工作几乎都是用数据挖掘方法来解决，但仍有不少研究偏重于人工分析作品语言风格。

作者分析问题衍生出很多更细的问题，主要有以下几个。

- *作者画像*：根据作品界定作者的年龄、性别或其他特性。例如，通过观察一个人讲英语

的方式，判断英语是否为他的母语。

- 作者验证：根据作者已有作品，推断其他作品是否也是他写的。拿法庭断案场景来讲更好理解，例如，分析嫌犯的写作风格（内容方面），以确定勒索信是不是他写的。
- 作者聚类：作者验证问题的延伸，用聚类分析方法把作品按照作者进行分类。

然而，作者分析研究领域最常见的还是作者归属问题，即如何从一群作者中找到作品的真正主人。

9.1.1 相关应用和使用场景

作者分析有很多应用场景，比如证实作者是谁，证实几本书有相同的作者，或者寻找社会媒体账号的主人。

从历史意义上来说，作者分析可用来核实某些文档是否真由公认的作者所写。有些作品的作者就存在争议，比如莎士比亚的几部戏剧，美国建国时期的联邦党人文集等。

单凭作者分析无法确定作者到底是谁，但是能够提供支持或反驳某一观点的依据。例如，在检验一首十四行诗是否是莎士比亚所写前，我们可以先分析他的若干部剧作，弄清楚他的写作风格。

作者分析问题最近几年也被用来确定社会媒体账号到底是谁在使用。例如，一个恶意用户可能在不同社交平台创建多个账号。把它们关联起来后，便于监管部门跟踪幕后的黑手——例如，当扰乱其他用户时。

举个陈旧点的例子，法庭上视作者分析为核心技术，主要靠它来提供文档是否出自嫌犯之手的证据。例如，嫌疑犯可能因为发骚扰邮件而被起诉。作者分析就能确定邮件是否是嫌犯所写。另一个在法庭中的使用场景是，解决作品归属纠纷。比如，遇到两个人就一本书到底是谁写的而纠缠不已的情况，法庭就可以借助作者分析技术找到作者可能是谁的证据。

作者分析也并不是绝对可靠的。最近研究发现，即使没有经过专业训练的人，让他们隐藏自己的写作风格，也会让作者归属问题变得困难无比。研究人员还探讨了模仿别人写作风格的问题，结果发现人们模仿得很像，作者分析的结果是这些伪作出自被模仿人之手。

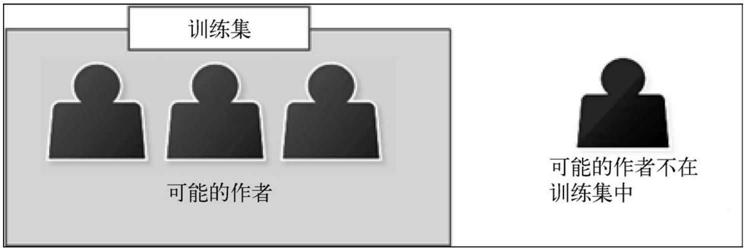
尽管存在很多问题，作者分析在越来越多的领域被证实十分有用，它是一个非常有意思的、值得研究的数据挖掘问题。

9.1.2 作者归属

作者归属可以看作是一种分类问题，已知一部分作者，数据集为多个作者的作品（训练集），目标是确定一组作者不详的作品（测试集）是谁写的。如果作者恰好是已知的作者里面的，这种问题叫作封闭问题。



如果作者可能不在里面，这种问题就叫作开放问题。不只是作者归属问题可以这样区分——任何数据挖掘应用，只要实际的类别可能不在训练集中的都叫作开放问题，对于这类问题进行挖掘，最终目标分两种情况，如果在训练集中，就返回找到的类别，如果不在，要给出不属于任何现有类别的提示。



在进行作者归属研究时，一般会受到两个限制。一是只能使用作品的内容，而不能使用写作时间、印刷形式、笔迹等信息。当然也有这样的做法，把这些信息整合到模型中，但一般来说这就不是作者归属而更像是数据融合问题。

第二个限制是不考虑作品的主题；相反，关注单词用法、标点和其他文本特征。原因在于，一个人如果是多面手，可以写多个主题的内容，作品的主题就不能反映作者的实际写作风格。关注主题关键字可能会导致过度拟合训练集——模型可能只是用同一个作者的同一个主题的作品进行训练。例如，根据我写的这本书为我的写作风格建模，可能会得出“数据挖掘”这个词能代表我的风格这样的结论，而事实上，我还写一些其他主题的内容。

接下来，用于作者归属问题的流水线跟第6章所用的很相似。首先，从文本中抽取特征；然后，对抽到的特征做进一步甄选；最后，训练算法分类器，预测文档的类别（作者）。

当然有些步骤跟第6章还是存在一些差别，主要集中在特征选取方面，后面会详细讲。我们还是先来划定待解决问题的范围。

9.1.3　获取数据

本章所用的图书数据集来自于古腾堡计划网站（www.gutenberg.org），该网站收集了大量版权失效的公版文学作品。实验用到以下作家的若干作品。

- 塔金顿（Booth Tarkington，22篇）
- 狄更斯（Charles Dickens，44篇）
- 伊迪斯·内斯比特（Edith Nesbit，10篇）
- 阿瑟·柯南·道尔（Arthur Conan Doyle，51篇）
- 马克·吐温（Mark Twain，29篇）
- 理查德·弗朗西斯·伯顿爵士（Sir Richard Francis Burton，11篇）
- 埃米尔·加博里奥（Emile Gaboriau，10篇）

以上总共是7位作家的177篇作品，文本数量还是相当可观的。作品名称列表、下载链接以及自动下载图书的代码请见本书配套的代码包。

用`requests`库下载作品文件到数据文件夹所在的目录。首先，在Data目录下创建文件夹存放这些作品。注意，下面代码指定的文件目录跟你实际创建的文件夹目录保持一致。

```
import os
import sys
data_folder = os.path.join(os.path.expanduser("~"), "Data", "books")
```

接着，就可以使用代码包中我写好的代码，从古腾堡计划的网站下载图书。下载完后将其保存到我们上面创建的目录中。

从代码包Chapter 9文件夹中找到getdata.py，保存到笔记本文件所在目录下，在新格子中输入以下代码。

```
!load getdata.py
```

在笔记本中，按下Shift+Enter运行该格子的代码。代码将会加载到格子中。然后再次点击代码，按下Shift+Enter运行加载的代码。代码要运行一段时间，运行完毕后会提示你。

大体翻看一下下载的作品，你会发现大部分都包含很多噪音——从数据分析的角度看至少是这样的。每篇作品前都有大段的声明文字，开始数据分析之前，得先把它们删掉。

9

我们可以直接从文件里将这些免责声明删除，但是丢失数据怎么办？我们可能会丢失先前做过的改动，从而导致实验结果无法重现，因此，不如在加载文件时跳过这部分——这样再次用这些文件做实验时，也能得到同样的结果（只要原文件没变）。代码如下：

```
def clean_book(document):
```

每篇作品前后各有一行文字，标识作品的开头和结尾，作品前后为古腾堡项目的说明。既然我们能根据这两行找到作品内容，因此，就按行来切分。

```
    lines = document.split("\n")
```

遍历文档的每一行，寻找作品的开头和结尾，中间部分就是作品内容。代码如下：

```
    start = 0
    end = len(lines)
    for i in range(len(lines)):
        line = lines[i]
        if line.startswith("*** START OF THIS PROJECT GUTENBERG"):
            start = i + 1
        elif line.startswith("*** END OF THIS PROJECT GUTENBERG"):
            end = i - 1
```

函数最后，用换行符把所有行再连接起来，得到作品内容。

```
    return "\n".join(lines[start:end])
```

现在，我们可以创建一个函数，加载所有图书，进行上述预处理操作，返回书的内容以及作家序号[①]。先来导入numpy。

```
import numpy as np
```

声明加载图书的函数，参数为图书所在目录books，该目录下是一系列以作者名字命名的子文件夹，图书文件就在这些子文件夹中。代码如下：

```
def load_books_data(folder=data_folder):
```

创建两个列表，分别用来存储文档和作者。

```
    documents = []
    authors = []
```

获取到books目录下所有的子文件夹。代码如下：

```
    subfolders = [subfolder for subfolder in os.listdir(folder)
                  if os.path.isdir(os.path.join(folder,
                     subfolder))]
```

遍历这些子文件夹，使用enumerate函数为这些子文件夹指定索引。

```
    for author_number, subfolder in enumerate(subfolders):
```

获取到子文件夹的绝对路径，查找里面的所有图书文件。

```
        full_subfolder_path = os.path.join(folder, subfolder)
        for document_name in os.listdir(full_subfolder_path):
```

对于找到的每一个图书文件，打开后读取里面的内容，对内容进行预处理后，将其添加到documents列表中。代码如下：

```
            with open(os.path.join(full_subfolder_path,
              document_name)) as inf:
                documents.append(clean_book(inf.read()))
```

① 用索引来表示，详见下文。——译者注

把分配给作家的索引号添加到authors列表中，其实authors就是类别列表。

```
            authors.append(author_number)
```

函数最后返回文档和类别（把类别列表转换为numpy数组）。

```
    return documents, np.array(authors, dtype='int')
```

调用上面定义的加载图书函数，就能获得图书文档及其它们的类别。

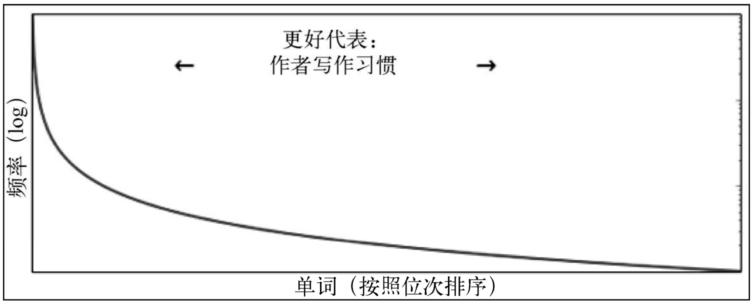
```
documents, classes = load_books_data(data_folder)
```

> 把这些数据加载到内存没有问题，因此，我们可以立即加载。如果数据集太大，内存装不下，较好的方法是每次只从一篇（或几篇）文档中抽取特征，把这些特征保存到文件或位于内存的矩阵中。

9.2 功能词

功能词是作者分析问题最早使用的一类特征，到现在来看，在词袋模型中使用功能词做特征效果依然不错。功能词指的是本身具有很少含义，却是组成（英语）句子必不可少的成分。例如，单词this和which的意思不是由它们本身的含义所决定，而是由它们在句子中充当的角色来决定。与功能词相对的是实词，比如tiger就有明确的含义，当人们在句子中看到这个单词时，脑海中就会浮现出大型猫科动物老虎的样子。

有些功能词的用法并不总是清晰、明确的。因此，从经验来看，选用使用频率较高的功能词做特征比较好（在所有文档中使用频率高，而不仅是在单个作者的作品中）。通常而言，使用越频繁的单词，对于作者分析能提供更多有价值的信息。相反，使用频率较低的单词，更适合用来做基于内容的文本挖掘，例如下章要讲的文档主题划分。



功能词的使用通常不是由文档内容而是由作者的使用习惯所决定，因此可以用它们来区分不同的作者。例如，很多美国人比较在意区分that和which在句子中的用法，澳大利亚等国家的人就不太在意。在使用这两个词的地方，有些澳大利亚人要么几乎无一例外地都用that，要么是which

用得多一点。这样的不同点，再加上成千上万个其他的微小差别，就形成了用于作者分析的模型。

9.2.1 统计功能词

我们可以使用第6章所用到的CountVectorizer统计功能词。我们把包含所有要查找的单词的词汇表（vocabulary）传递进去，如果没有传词汇表（第6章就没有），它会从数据集中学习。所有单词都在训练集中（取决于其他参数）。

首先，创建功能词词汇表，用列表存储。至于确切来说哪些是功能词，哪些不是，有待商榷。我从已发表的研究成果中找到下面这些功能词，它们还是比较可靠的。

```
function_words = ["a", "able", "aboard", "about", "above", "absent",
"according" , "accordingly", "across", "after", "against",
"ahead", "albeit", "all", "along", "alongside", "although",
"am", "amid", "amidst", "among", "amongst", "amount", "an",
"and", "another", "anti", "any", "anybody", "anyone",
"anything", "are", "around", "as", "aside", "astraddle",
"astride", "at", "away", "bar", "barring", "be", "because",
"been", "before", "behind", "being", "below", "beneath",
"beside", "besides", "better", "between", "beyond", "bit",
"both", "but", "by", "can", "certain", "circa", "close",
"concerning", "consequently", "considering", "could",
"couple", "dare", "deal", "despite", "down", "due", "during",
"each", "eight", "eighth", "either", "enough", "every",
"everybody", "everyone", "everything", "except", "excepting",
"excluding", "failing", "few", "fewer", "fifth", "first",
"five", "following", "for", "four", "fourth", "from", "front",
"given", "good", "great", "had", "half", "have", "he",
"heaps", "hence", "her", "hers", "herself", "him", "himself",
"his", "however", "i", "if", "in", "including", "inside",
"instead", "into", "is", "it", "its", "itself", "keeping",
"lack", "less", "like", "little", "loads", "lots", "majority",
"many", "masses", "may", "me", "might", "mine", "minority",
"minus", "more", "most", "much", "must", "my", "myself",
"near", "need", "neither", "nevertheless", "next", "nine",
"ninth", "no", "nobody", "none", "nor", "nothing",
"notwithstanding", "number", "numbers", "of", "off", "on",
"once", "one", "onto", "opposite", "or", "other", "ought",
"our", "ours", "ourselves", "out", "outside", "over", "part",
"past", "pending", "per", "pertaining", "place", "plenty",
"plethora", "plus", "quantities", "quantity", "quarter",
"regarding", "remainder", "respecting", "rest", "round",
"save", "saving", "second", "seven", "seventh", "several",
"shall", "she", "should", "similar", "since", "six", "sixth",
"so", "some", "somebody", "someone", "something", "spite",
"such", "ten", "tenth", "than", "thanks", "that", "the",
"their", "theirs", "them", "themselves", "then", "thence",
"therefore", "these", "they", "third", "this", "those",
"though", "three", "through", "throughout", "thru", "thus",
"till", "time", "to", "tons", "top", "toward", "towards",
```

```
"two", "under", "underneath", "unless", "unlike", "until",
"unto", "up", "upon", "us", "used", "various", "versus",
"via", "view", "wanting", "was", "we", "were", "what",
"whatever", "when", "whenever", "where", "whereas",
"wherever", "whether", "which", "whichever", "while",
"whilst", "who", "whoever", "whole", "whom", "whomever",
"whose", "will", "with", "within", "without", "would", "yet",
"you", "your", "yours", "yourself", "yourselves"]
```

既然有了功能词列表，我们就来创建功能词统计工具。后面，会把它加到流水线中。

```
from sklearn.feature_extraction.text import CountVectorizer
extractor = CountVectorizer(vocabulary=function_words)
```

9.2.2 用功能词进行分类

接下来，导入所需的几个类，唯一的新内容支持向量机在下节会讲（现在把它看作是标准的分类算法即可）。导入用支持向量机算法进行分类的SVC类，以及其他一些我们用过的标准工作流工具。

```
from sklearn.svm import SVC
from sklearn.cross_validation import cross_val_score
from sklearn.pipeline import Pipeline
from sklearn import grid_search
```

支持向量机接收一系列参数。现阶段照我设置的参数来就行，下节再深入探讨参数值选择。我们用字典结构来组织参数。参数kernel使用linear和rbf。C的值取1或10（参数说明请见下节）。接着用网络搜索法寻找最优参数值。

```
parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
svr = SVC()
grid = grid_search.GridSearchCV(svr, parameters)
```



> 高斯内核（例如rbf）只适用于数据集相对较小的情况，比如特征数少于10 000。

9

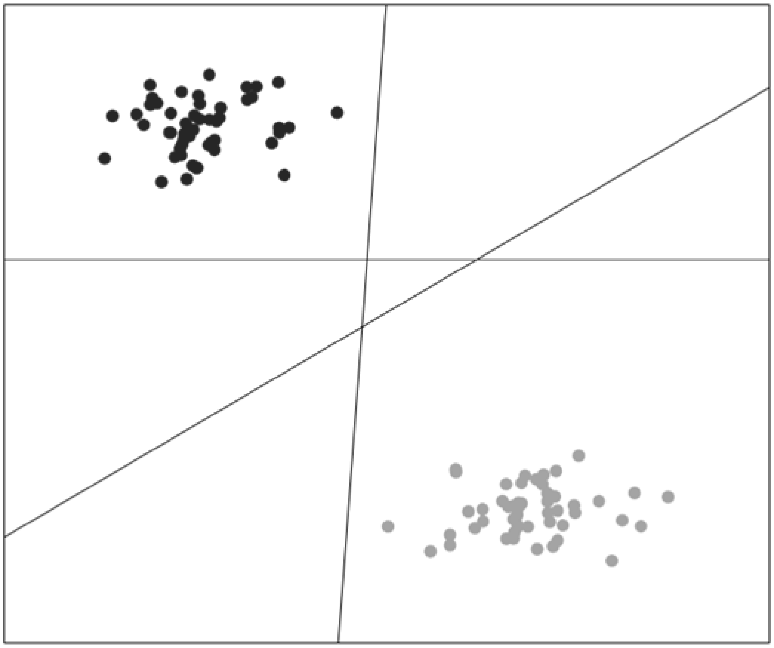
接着，创建流水线，把特征抽取和参数搜索两个步骤加入到流水线中，特征（仅功能词）抽取使用CountVectorizer类，参数搜索使用SVM。代码如下：

```
pipeline1 = Pipeline([('feature_extraction', extractor),
                      ('clf', grid)
                      ])
```

然后，使用cross_val_score对该流水线的结果进行交叉检验，正确率为0.811，大约80%的预测结果正确。对于只有7个作者来说，这个结果很好！

9.3 支持向量机

支持向量机（SVM）分类算法背后的思想很简单，它是一种二类分类器（扩展后可用来对多个类别进行分类）。假如我们有两个类别的数据，而这两个类别恰好能被一条线分开，线上所有点为一类，线下所有点属于另一类。SVM要做的就是找到这条线，用它来做预测，跟线性回归原理很像。只是SVM要找出最佳的分割线。下图中有三条线，分别为蓝色、黑色和绿色线，都能把两类数据区分开来。你说哪条分类效果更好呢？



凭直觉，人们往往会选择从左下到右上的这条线[①]，它把两类数据更好地分开了。也就是说，数据集中的每个点到这条线的距离都是最远的。

找到这样的一条线其实是最优化问题，也就是要让各点到分割线之间的距离最大化。

> 虽然最优化问题的公式推导不在本书讲述范围之列，我还是建议感兴趣的读者自行查看http://en.wikibooks.org/wiki/Support_Vector_Machines，了解详细步骤。此外，你还可以访问http://docs.opencv.org/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html，了解支持向量机相关内容。

① 如果你查看原书PDF文件的话，这条线指的是图中的蓝线。PDF文件获取方法请见前言部分。建议读者下载查看。彩色图毕竟比黑白图形象。——译者注

9.3.1 用 SVM 分类

模型训练完毕，就能找到使得两类之间间隔最大的一条线。新数据点分类问题就简化为：数据点在线上还是线下？如果在线上，就被分到线上那一类；如果位于线下，自然属于线下那一类。

对于多种类别的分类问题，我们创建多个SVM分类器——每个还是二类分类器。连接多个分类器有不少方法，从中任选一种即可。最简单的方法是为每个类别创建一对多（one-versus-all）分类器，把训练数据分为两个类别——属于特定类别的数据和其他所有类别数据。对新数据进行分类时，从这些类别中找出最匹配的。大多数SVM多类分类器都能自动完成上述过程。

前面代码中有两个参数：`C`和`kernel`。`kernel`参数下节讲，`C`参数对于训练SVM来说很重要，我们现在来讲下。`C`参数与分类器正确分类比例相关，但可能带来过拟合的风险。`C`值越高，间隔越小，表示要尽可能把所有数据正确分类。`C`值越小，间隔越大——有些数据将无法正确分类。`C`值低，过拟合训练数据的可能性就低，但是分类效果可能会相对较差。

SVM（基础形式）局限性之一就是只能用来对线性可分的数据进行分类。如果线性不可分呢？这时我们就要用到内核函数。

9.3.2 内核

如果数据线性不可分，就需要将其置入更高维的空间中，加入更多伪特征直到数据线性可分（如果你加入足够多恰当的特征，总能把数据分开）。

寻找最佳分隔线时往往需要计算个体之间的内积[①]。对于使用点积（dot product）的函数，我们可以创建新特征而无需实际定义这些特征。因为我们不知道这些特征到底是什么样子，所以用点积很方便。我们把内核函数定义为数据集中两个个体函数的点积，而不是使用个体自身（及构造的特征）。

接下来就可以计算点积（或近似值），然后就能使用它。

常用的内核函数有几种。线性内核最简单，它无外乎两个个体的特征向量的点积、带权重的特征和偏置项。多项式内核提高点积的阶数（比如2）。此外，还有高斯内核（rbf）、Sigmoind内核。前面例子中，我们比较了线性内核和`rbf`内核的效果。

这些内核能够有效地确定两类数据之间的距离，SVM可以据此对新数据进行分类。理论上讲，任何距离度量方法都可以用，但是在训练SVM时，最优化问题的复杂程度会有所不同。

`scikit-learn`实现的SVM，可以通过`kernel`参数指定内核函数，正如我们在前面例子中见过的。

① inner product，也称作点积。——译者注

9.4　字符 N 元语法

我们前面研究的是如何用功能词做特征预测文档的作者。接下来看下字符N元语法特征。N元语法由一系列的N个为一组的对象组成。N为每组对象的个数（对于文本来说，N通常取2到6之间的值）。基于单词的N元语法被广泛用在通常与文档主题相关的各项研究中。然而，基于字符的N元语法被证明在作者归属问题上效果很好。

我们可以把文档看成是由一系列字符组成的，从里面抽取N元语法，训练得到模型。常用模型有几个，其中标准模型跟我们之前用到的词袋模型很相似。

分别为训练语料中各个不同的N元语法创建一个特征。例如，由字母e、空格和字母t组成的N元语法`<e t>`（尖括号标识N元语法的边界，不是N元语法的内容）。然后，我们使用N元语法在训练集中的频率训练模型，再用生成的特征矩阵训练分类器。

> 基于字符的N元语法有几种不同的定义方法。比如，有些应用只选取单词内的字符，忽略空格和标点。但有些就使用这些信息（比如我们这一章）。

关于基于字符的N元语法为什么效果比较好，广为接受的理论是，人们在写作时，往往选取那些他们讲起来很容易的单词，而字符N元模型（至少N取 2到6之间时）跟这些音素具有很好的相似关系——音素指的是我们讲话时，组成单词发音的那些声音。从这种意义上讲，用字符N元模型可以很好地模拟讲话时经常用到的单词，而这些单词形成了写作风格。这是创建新特征的通用模式：先找到哪些概念影响最终结果（写作风格）的理论依据，然后创建跟这些概念相近或是能够量化这些概念的特征。

字符N元语法矩阵的一个主要特点是数据稀疏，随着N值的增加，稀疏程度逐渐加大，且速度很快。N取2时，我们的特征矩阵中大约75%的项都是0，N取5时，93%以上的项为0。比起基于词语的N元语法矩阵来说，稀疏程度要低一些，因此，适用于单词分类的分类器处理数据，不会有太大问题。

抽取字符 N 元语法

我们接下来用`CountVectorizer`类来抽取N元语法，需要设置`analyzer`参数，指定N的值。

`scikit-learn`的N元语法抽取工具提供了`range`参数，允许用户同时抽取不同长度的N元语法。我们这里用不到，只抽取相同长度即可，`range`参数的两个值使用相同值即可。要抽取长度为3的N元语法，`range`的值需要设置为(3, 3)。

我们可以重用前面网格搜索代码，但是需要在新流水线中指定新的特征抽取器。

```
pipeline = Pipeline([('feature_extraction', CountVectorizer(analyzer='
char', ngram_range=(3, 3))),
                     ('classifier', grid)
                     ])
scores = cross_val_score(pipeline, documents, classes, scoring='f1')
print("Score: {:.3f}".format(np.mean(scores)))
```

> 功能词和字符N元模型之间存在大量隐式重合，因为字符序列更可能出现在功能词中。然而，实际上两者差别很大，字符N元模型能够捕获标点的使用特点，功能词自然做不到。例如，字符N元语法能够包含句号，而基于功能词的方法就只能使用句号前的单词。

9.5 使用安然公司数据集

安然公司（Enron）是20世纪90年代末期世界上最大的能源公司之一，年收入高达1000亿美元。2000年时，拥有20 000余名员工，看不出公司有什么严重问题。

2001年，安然丑闻发生，调查人员发现安然为谋取暴利，在全公司范围内账务存在系统性造假现象。丑闻被揭发后，安然公司的股价从2000年的90多美元一下子跌到了2001年的1美元。安然随即申请破产保护，留下的残局，五年多以后才收拾干净。

作为对安然公司调查的一部分，美国联邦能源署公开了60多万封电子邮件。从那时起，这些数据就被广泛应用于社会媒体分析、欺诈分析等众多问题的研究。这些数据用于作者归属问题研究也不错，因为我们能获得发件人及他们写的邮件，语料规模比之前见过的很多数据集都要大得多。

9.5.1 获取安然数据集

安然公司的邮件可以从卡内基梅隆大学的网站下载：https://www.cs.cmu.edu/~./enron/。

> 整个数据集为423MB，压缩格式为gzip。如果你用的不是Linux系统，可以使用免费的7-zip（(http://www.7-zip.org/）等软件来解压。

下载电子邮件语料，将其解压到Data目录下。解压后的目录默认为enron_mail_20110402。

因为要做作者归属分析，我们只需要那些明确知道发件人是谁的邮件。因此，查看每位用户的发件夹——那里面存放的是他们所发的邮件。

在笔记本中，指定安然数据集所在的位置。

```
enron_data_folder = os.path.join(os.path.expanduser("~"), "Data",
  "enron_mail_20110402", "maildir")
```

9.5.2　创建数据集加载工具

我们现在就来创建一个函数，它接收几个发件人作为参数，返回他们所发送的邮件。我们需要的有效信息是邮件内容而不是邮件本身。因此，还需要邮件解析器。代码如下：

```
from email.parser import Parser
p = Parser()
```

我们后面会用到该解析器从邮件中抽取邮件内容。

因为是随机选取收件人，为了重现实验结果，我们设置随机状态。

```
from sklearn.utils import check_random_state
```

数据加载函数提供几个参数，大部分是为了确保得到的数据集类别分布相对比较均衡。有些用户发了成千上万封邮件，而有的用户只发了几十封。我们用`min_docs_author`参数指定每个发件人至少发过10封邮件，用`max_docs_author`参数指定最多从一个用户那里抽取100封邮件。我们还用`num_authors`限定了收件人数量——默认为10。代码如下：

```
def get_enron_corpus(num_authors=10, data_folder=data_folder,
                     min_docs_author=10, max_docs_author=100,
                     random_state=None):
    random_state = check_random_state(random_state)
```

接下来，获取到安然公司员工的邮箱，邮箱其实就是data_folder文件夹中各子文件夹的名称。随机对得到的邮箱列表进行排序。只要随机状态相同，实验结果就可以重现。

```
    email_addresses = sorted(os.listdir(data_folder))
    random_state.shuffle(email_addresses)
```

> 你可能很好奇我们既然后面接着要随机调整顺序，那为什么还要事先进行排序。因为`os.listdir`函数每次返回结果不一定相同，在使用该函数前先排序，从而保持返回结果的一致性。然后我们用随机状态的`shuffle`函数随机选取一组收件人。如果需要的话，随机状态函数可以返回跟之前相同的结果。

然后，我们创建文档列表、类别列表，`author_num`指的是每个新发件人的类别编号。这次我们不用`enumerate`函数，因为有些发件人我们用不到。例如，发过不够10封邮件的，就不会用。代码如下：

```
    documents = []
    classes = []
    author_num = 0
```

我们还需要记录我们所用到的收件人及他们的编号。数据挖掘过程用不到，但是在可视化时能用到，便于我们确定收件人。authors字典用于将用户名和类别关联起来。代码如下：

```
authors = {}
```

接下来，遍历邮箱文件夹，查找它下面名字中含有“sent”的表示发件箱的子文件夹。代码如下：

```
for user in email_addresses:
  users_email_folder = os.path.join(data_folder, user)
  mail_folders = [os.path.join(users_email_folder,
    subfolder) for subfolder in os.listdir(users_email_folder)
                    if "sent" in subfolder]
```

获得子文件夹中的每一封邮件。我们用到了try-except语句，因为有些用户的发件箱目录还有子目录。要获取目录层次更深的邮件，我们的代码还需要做些改进，现在我们先跳过这些用户。代码如下：

```
try:
  authored_emails = [open(os.path.join(mail_folder,
    email_filename), encoding='cp1252').read()
  for mail_folder in mail_folders
  for email_filename in os.listdir(mail_folder)]
except IsADirectoryError:
    continue
```

接下来，检测是否获取到了至少10封邮件（或min_docs_author指定的其他值）。

```
if len(authored_emails) < min_docs_author:
    continue
```

下一步，如果发件人发了大量邮件，只取前100封（具体数量由max_docs_author指定）。

```
if len(authored_emails) > max_docs_author:
    authored_emails = authored_emails[:max_docs_author]
```

解析邮件，获取邮件内容。我们对邮件头部不感兴趣——发件人所能改动的内容比较少，因此对作者分析没有多大用处。然后把邮件内容添加到数据集中。

```
contents = [p.parsestr(email)._payload for email in
  authored_emails]
documents.extend(contents)
```

把该发件人添加到类别列表中，每一封邮件添加一次。

```
classes.extend([author_num] * len(authored_emails))
```

记录该收件人的编号，再把编号加1，以便下一个收件人使用。

```
authors[user] = author_num
author_num += 1
```

检测收件人数量是否达到我们设置的值，如果是，跳出循环，返回数据集。

```
        if author_num >= num_authors or author_num >=
          len(email_addresses):
            break
```

返回邮件数据集和类别以及收件人字典。

```
    return documents, np.array(classes), authors
```

在上述函数的外面，调用这个函数，我们就能得到数据集。我们使用随机状态14（全书都是），但是你可以尝试其他值或者将其设置为none，这样每次调用这个函数时将随机获取一组数据。

```
documents, classes, authors = get_enron_corpus(data_folder=enron_data_
folder, random_state=14)
```

如果你看下现有的数据集，你就会发现还需要做进一步的预处理。数据集有不少噪音，但最致命的问题（从数据分析角度看）是，它还包含其他用户所写的内容，回复邮件时会带上别人之前写的邮件。我们来看下邮件`documents[100]`：

I am disappointed on the timing but I understand. Thanks. Mark

-----Original Message-----

From: Greenberg, Mark

Sent: Friday, September 28, 2001 4:19 PM

To: Haedicke, Mark E.

Subject: Web Site

Mark -

FYI - I have attached below a screen shot of the proposed new look and feel for the site. We have a couple of tweaks to make, but I believe this is a much cleaner look than what we have now.

在这篇文档中，上面的邮件是对下面邮件的回复，这种情况很常见。第一部分是来自Mark Haedicke，而下面的邮件是Mark Greenberg写给Mark Haedicke。只有前面的内容（第一处“-----Original Message-----”之前）是发件人所写，这才是我们所关注的。

抽取这些信息不容易。邮件没有统一的格式。不同的邮件服务提供商使用不同的头部，对于回复内容有不同的定义形式，简直就是“为所欲为”[1]。就是在这样的环境中，电子邮件还被广泛使用，简直就是一个奇迹。

① 读到此处，不禁想起各种浏览器的纷争。——译者注

当然还是有些共用的模式可以用。我们可以用quotequail包来查找邮件中的新内容，抛弃被回复的邮件及其他不相干信息。

> 你可以使用pip安装quotequail：pip3 install quotequail。

我们编写一个简单的函数封装quotequail的功能，方便调用它处理所有的文档。首先导入quotequail，声明函数。

```
import quotequail
def remove_replies(email_contents):
```

接着用quotequail把邮件解析为几个部分，返回字典结构。代码如下：

```
    r = quotequail.unwrap(email_contents)
```

有时候，r可能为none，比如邮件出于这样那样的原因解析失败。遇到这种情况，邮件原样返回。在处理真实数据集时，经常需要检测变量值是否为空。代码如下：

```
    if r is None:
        return email_contents
```

quotequail的返回结果中，我们真正需要的是text_top这部分。如果字典r中存在text_top，返回它的值。

```
    if 'text_top' in r:
        return r['text_top']
```

如果不存在，也就是quotequail没能找到该键。它也可能找到了text键，那返回text的值。代码如下：

```
    elif 'text' in r:
        return r['text']
```


最后，如果这两个键都没有找到，返回邮件全部内容，希望多少会对数据分析有点用处。

```
    return email_contents
```

对数据集中的每一封邮件，运行上述函数，做一遍预处理。

```
documents = [remove_replies(document) for document in documents]
```

我们之前看过的那封含有被回复邮件的文档，经过处理后，只包含Mark Greenberg所发的内容。

I am disappointed on the timing but I understand. Thanks. Mark

9.5.3　组装起来

我们可以使用前面实验所用到的参数空间和分类器——在新数据集上进行训练。默认情况下，scikit-learn会重新进行训练——后续调用fit()函数将会丢掉之前的信息。

> 还有一类算法叫作线上学习，它们使用新数据更新训练结果，但不是每次都重新进行训练。后续章节包括第10章新闻语料分类，我们会实际用到线上学习这类算法。

跟前面一样，我们使用cross_val_score计算正确率并输出结果。代码如下：

```
scores = cross_val_score(pipeline, documents, classes, scoring='f1')
print("Score: {:.3f}".format(np.mean(scores)))
```

F1值为0.523，对于包含这么多噪音的数据集来说，结果还算可以。添加更多数据（比如增加max_docs_author的值）应该会提升效果。

9.5.4　评估

一般来说，只凭借一个数字来评估效果不是个好主意，可能会忽略很多东西。拿F值来说，它考察的点就比较全面，那些分类结果好，却没有实际用处的雕虫小技也就不能蒙混过关。前面多次使用的正确率就有破绽，比如邮件过滤器把所有邮件归为垃圾邮件，也能达到80%的正确率，但是这种方法却没有实际用处。出于这个原因，我们还要对评估结果进行深入探讨。

我们先来看下混淆矩阵，第8章用神经网络破解验证码曾经用过。首先，对测试集数据进行预测。前面代码使用cross_val_score，并没有给出可用的训练模型，因此我们需要重新训练一个模型。我们先来把语料切分为训练集和测试集。

```
from sklearn.cross_validation import train_test_split
training_documents, testing_documents, y_train, y_test =
train_test_split(documents, classes, random_state=14)
```

接着，把训练集传入流水线，进行训练，接着对测试集进行预测。

```
pipeline.fit(training_documents, y_train)
y_pred = pipeline.predict(testing_documents)
```

你可能想知道最好的参数组合是什么。我们可以方便地从网格搜索对象（流水线classifier这一步）中获取到这些参数组合。

```
print(pipeline.named_steps['classifier'].best_params_)
```

上述代码输出分类器的所有参数。然而大部分参数都使用默认值。只有C和kernel两个参数，我们使用网格搜索寻找合适的值，它们分别被设置为1和linear。

创建混淆矩阵：

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_pred, y_test)
cm = cm / cm.astype(np.float).sum(axis=1)
```

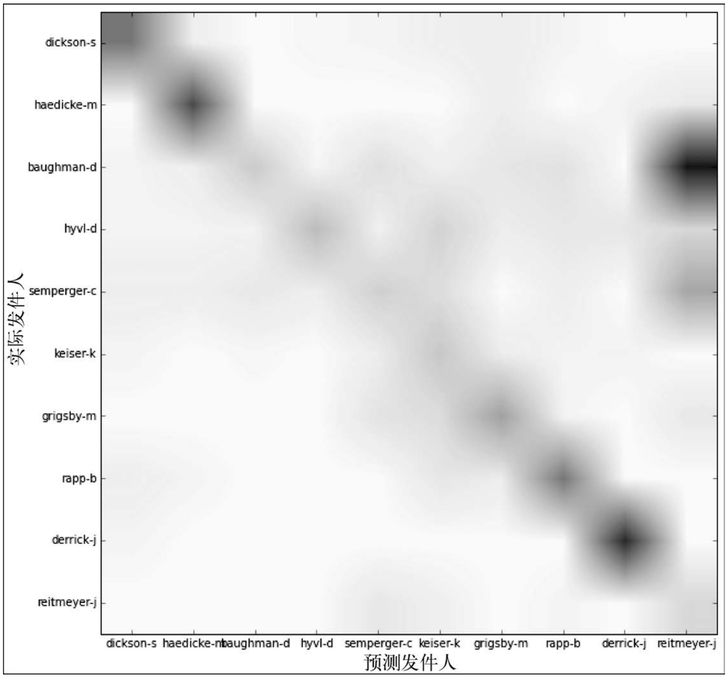
我们接下来作图要用到刻度，因此需要取到发件人。还好，加载数据集时，我们用字典来保存了发件人及其编号。代码如下：

```
sorted_authors = sorted(authors.keys(), key=lambda x:authors[x])
```

最后，使用matplotlib绘制混淆矩阵。代码跟上章大同小异，不同的地方用粗体表示，只是把坐标轴刻度改为发件人。

```
%matplotlib inline
from matplotlib import pyplot as plt
plt.figure(figsize=(10,10))
plt.imshow(cm, cmap='Blues')
tick_marks = np.arange(len(sorted_authors))
plt.xticks(tick_marks, sorted_authors)
plt.yticks(tick_marks, sorted_authors)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

图像如下。



我们可以看到哪些发件人在大多数情况下都能正确预测——正确率高的形成了一条清晰的对角线。我们还能看出哪些发件人的邮件经常被弄混（颜色越深，弄混的次数越多）：例如，baughman-d的邮件经常被当作是reitmeyer-j发的。

9.6　小结

本章研究的是文本挖掘领域中的作者归属问题，我们分析了两类特征的效果：功能词和字符N元语法。我们在词袋模型中使用功能词——提前选好的一组词，计算出这些词的词频。字符N元语法的处理流程跟基于单词的N元语法很相似，但是分析器转而关注字符而不是单词。此外，N元语法由一系列字符组成。单词N元语法由于可以提供单词的语境而没有增加多少计算量，在某些应用中也值得一试。

我们用SVM来进行分类，它通过找出使得两个类别之间间隔最大的线进行分类。线上的属于一类，线下的属于另外一类。跟其他分类任务一样，我们也需要有数据集（邮件）。

我们使用安然公司的邮件作为数据集，它包含很多噪音，比如包含被回复的邮件等。所以分类效果比起图书数据集要差不少。然而，在有10个发件人的情况下，半数以上邮件的发件人都能预测正确。

下一章，我们要研究的问题是没有目标类别的情况下，怎么对数据进行分类。这种叫作无监督学习，比起预测更像是探索性问题。我们要使用的依然是充满噪音的文本数据集。准备好接受挑战了吗？

第 10 章 新闻语料分类

前面大部分章节在对数据进行挖掘前，新数据所属的目标类别范围是已知的。在训练过程中，我们能够了解到变量跟类别之间存在怎样的关系。这种已知目标类别的学习任务，叫作有监督学习。本章研究的是在不知道类别的情况下如何进行数据挖掘。这叫作无监督学习，它偏重于探索、发现隐藏在数据里的信息，而不是用模型来分类，其探索性更强。

本章将介绍如何对新闻语料进行聚类，以发现其中的趋势和主题。本章还将讨论怎样从链接聚合网站抽取新闻语料。

本章主要介绍如下几个概念。

- 从任意新闻网站获取文本内容
- 使用reddit网站API采集有趣的新闻报道
- 使用聚类分析进行无监督的数据挖掘
- 抽取文档主题
- 用线上学习方法无需再次训练即可更新模型
- 组合不同的模型进行聚类

10.1 获取新闻文章

本章将构建一个按照主题为最新的新闻报道分组的系统。你可以运行几周（或更长时间）以了解这段时间新闻趋势的变化。

系统首先从流行的链接聚合网站reddit[①]寻找新闻报道的链接。reddit存储了大量其他网站的链接，还提供讨论区。网站收集的链接按照类别进行分类，这些类别被统称为subreddit，比如有电

① Y Combinator孵化的项目。最初使用Lisp语言开发，后改用Python，网站开放源代码，可以从Github上找到。创始人Steve Huffman在Udacity上开设了一门讲解Web开发的课程，课程编号为CS253。Steve卖掉reddit后，创立了专注于旅游搜索的Hipmunk网站。2015年再度回到reddit，出任CEO。web.py的作者Aaron Swartz也是reddit的联合创始人。——译者注

视节目、趣味图片等类别。本章关注的是新闻这一类链接。本章使用`/r/worldnews`类别的链接，但是我们所编写的代码也可以用来抓取其他类别的语料。

本章的目标是下载大家所关注的新闻报道，对其进行聚类分析，寻找其中的主题或主要概念，无需人工分析成百上千篇新闻报道，就能洞察人们关注的焦点。

10.1.1　使用 Web API 获取数据

前面有几章都是使用Web API从网站抽取数据。例如，第7章就用到Twitter网站的API。采集数据是数据挖掘流水线至关重要的一个环节。Web API是采集多个主题数据的绝佳工具。

使用Web API采集数据，有三个注意事项：授权方法、请求频率限制和API端点（endpoints）。

*授权方法*是数据提供方用来管理数据采集方的。数据提供方以此了解谁正在采集数据，确保采集方抓取数据的频率没有超出上限，同时对采集方都采集了哪些数据也做到心中有数。对于大多数网站，普通的个人用户账号就能用来采集数据，但也有部分网站要求采集方使用正式的开发者账号。

*采集频率限制*规定了采集方在约定时间内的最大请求次数，尤其是针对免费提供的服务。在使用数据获取接口时，一定要了解清楚，不同的网站很可能有着不同的规定。Twitter的API要求每15分钟（视调用的API而定）之内最多请求180次数据。reddit规定每分钟至多发起30次请求，下面我们还会讲到。其他有些网站会限制每天的请求次数，当然也有限制到秒级别的。即使同一个网站，不同的API调用，也有着截然不同的采集频率限制。例如，谷歌地图对每种资源API的调用限制，与按小时请求就有着不同规定，对前者限制更为严格。

> *如果你的应用或实验需要发起更多的请求和更快的响应，超出了免费API所能提供的，你可以寻求与API提供商进行商业合作。*

*API端点*指的是用来抽取信息的实际网址。不同网站提供不同的接口，大部分Web API提供RESTful接口（Representational State Transfer，表述性状态转移）。RESTful接口通常与HTTP协议使用相同的操作：GET、POST、DELETE是最常用的。例如，使用下述API端点检索某一资源的相关信息：www.dataprovider.com/api/resource_type/resource_id/。

获取信息，只需要发送HTTP GET请求到指定的网址。服务器返回资源信息、信息类型和ID。大多数API都是按照这种结构进行封装的，虽然在具体实现上会有所差异。大多数提供API的网站，都会给出详细的文档，列出所有开放使用的API的具体参数。

我们来看下获取信息的具体步骤。首先，设置连接reddit网站所用到的参数，这里需要用到reddit开发者密钥。从https://www.reddit.com/login登录reddit网站，打开https://www.reddit.com/prefs/apps。点击“are you a developer? create an app…”，填写表单，选择`script`类型。你就能得到用

户ID和密钥，新建一个笔记本文件，输入以下代码。

```
CLIENT_ID = "<Enter your Client ID here>"
CLIENT_SECRET = "<Enter your Client Secret here>"
```

reddit还要求你在使用他们的API时，设置唯一的用户代理（user agent）字符串。在设置用户代理时，注意把自己的reddit用户名也写上，以创建唯一标识你应用的用户代理字符串。我的用户代理字符串中包含本书的英文名、章节编号“chapter 10”及版本号“0.1”，你可以使用其他内容。如果你的用户代理跟别人重复，你的请求频率将受到很大限制。

```
USER_AGENT = "python:<your unique user agent> (by /u/<your reddit
username>)"
```

此外，你需要使用自己的用户名和密码登录reddit。如果你还没有账号，请注册一个（免费且无需验证个人信息）。

> 下一步会用到你的密码，把代码分享给别人之前，记得删除它。如果你不打算在代码中写入密码，把密码设置为none，这样每次运行时输入密码即可。但是，你需要在启动笔记本文件的终端输入，而不是在笔记本文件里输入，这是由笔记本文件的工作方式决定的。如果你无法在终端输入，请直接在代码里设置好密码。IPython的开发人员正在编写修复这个问题的插件，但是写作本书时，插件还没写好。

指定用户名和密码，代码如下：

```
USERNAME = "<your reddit username>"
PASSWORD = "<your reddit password>"
```

接着，创建登录函数，使用以上信息进行登录。reddit的登录接口将会返回进一步连接所需要的令牌，这也就是登录函数的返回结果。函数声明如下：

```
def login(username, password):
```

首先，如果你不想把密码写到代码里，把它设置为none，但是你需要在命令行输入，前面已经说过。代码如下：

```
    if password is None:
        password = getpass.getpass("Enter reddit password for user {}:
".format(username))
```

使用独一无二的用户代理很重要，否则你的连接将严重受限。代码如下：

```
    headers = {"User-Agent": USER_AGENT}
```

接着，创建HTTP授权对象，以登录reddit服务器。

```
    client_auth = requests.auth.HTTPBasicAuth(CLIENT_ID, CLIENT_
SECRET)
```

发起POST请求到`access_token`端点以实现登录，POST的数据有用户名、密码、授权类型。本例中授权类型为使用密码进行登录。

```
    post_data = {"grant_type": "password", "username": username,
"password": password}
```

最后，函数使用`requests`库发起登录请求（通过HTTP POST请求实现），POST请求返回的结果为字典类型。其中有一项就是进一步请求所需的令牌。代码如下：

```
    response = requests.post("https://www.reddit.com/api/v1/access_
token", auth=client_auth, data=post_data, headers=headers)
    return response.json()
```

调用上述登录函数，获取令牌。

```
token = login(USERNAME, PASSWORD)
```

令牌对象为一字典结构，其中的`access_token`键对应的值就是进一步请求所需的令牌。该对象还包含令牌的使用范围（全局）和过期时间。例如：

```
{'access_token': '<semi-random string>',  'expires_in': 3600,
'scope': '*',  'token_type': 'bearer'}
```

10.1.2　数据资源宝库 reddit

链接聚合网站reddit（www.reddit.com）拥有几亿用户，虽然英文版主要面向美国。每个用户都可以发布他们感兴趣的网站的链接，同时为该链接指定标题。其他用户对其点赞，表示他们喜欢这个链接的内容，也可以投反对票，表示不喜欢。点赞最高的链接将被移到网页的最上面，没有多少人喜欢的将不会显示。随着时间的推移，先前发表的链接也将不再显示（根据喜欢数来定）。分享的链接被点赞后，用户将会获得叫作karma[①]的积分，这也是为了激励用户只分享好故事。

reddit用户也可以发表链接之外的其他内容，这些内容叫作自己的广播[②]，它由用户输入的标题和正文组成，通常是为了提问或是引发讨论，不在积分赚取范围之内。本章仅关注链接类广播，不关注评论类的。

网站的所有广播被分到叫作subreddit的不同栏目，每个栏目包含一系列与之相关的广播。用户发布广播时，需要选择广播所属的栏目。每个栏目都有自己的管理员和内容管理规则，用户不能发表与栏目无关的内容。

① karma，指因果报应。reddit网站使用该词，应是取其善有善报之意。——译者注
② 广播对应英文单词post，指的是用户发表的一条内容。——译者注

广播默认根据热度（Hot）进行排序，一条广播的热度由其存在的时间、喜欢数、不喜欢的次数来决定。除了热度外，还有最新发表的（New）和一定时间内最受欢迎的（Top）两种排序方法，最新广播可能包含大量的垃圾信息。本章将按照热度来选取广播，这些广播比较新，内容质量也比较高（单纯根据时间排序，得到的链接有很多是垃圾链接）。

使用我们前面获取到的令牌，就可以获取一个栏目的一系列链接。`/r/<subredditname>` API端点默认返回所指定栏目的热门文章。我们把栏目指定为世界新闻`/r/worldnews`。

```
subreddit = "worldnews"
```

我们使用字符串格式化方法，用刚指定的栏目来创建完整的URL。

```
url = "https://oauth.reddit.com/r/{}".format(subreddit)
```

接下来，需要设置头部，这样才能指定授权令牌，设置独一无二的用户代理，争取不会被过分限制请求次数。代码如下：

```
headers = {"Authorization": "bearer {}".format(token['access_token']),
           "User-Agent": USER_AGENT}
```

然后，跟之前一样，使用`requests`库发起请求，别忘了指定头部。

```
response = requests.get(url, headers=headers)
```

调用response的`json`方法，将会得到包含reddit返回信息的一个Python字典。它包含给定栏目的25条广播，对它们进行遍历，输出每条广播的标题。广播的相关内容存储在字典键为data的那一项中。代码如下：

```
for story in result['data']['children']:
    print(story['data']['title'])
```

10.1.3　获取数据

我们数据集的每条广播都来自`/r/worldnews`栏目的热度列表。上节讲解了连接reddit网站和抽取广播内容的方法。我们来创建一个函数，把这两个步骤整合起来，抽取指定栏目每条广播的标题、链接和喜欢数。

我们遍历世界新闻栏目，最多一次获取100篇新闻报道。我们还可以使用分页，reddit最多允许我们读多少页，我们就读多少页。但是我们这里最多读5页。

因为我们代码会重复调用同一个API，为了防止超出网站所规定的采集频率限制，我们可以使用`sleep`函数，先来导入它。

```
from time import sleep
```

接下来要定义的这个函数接收三个参数，分别为栏目名称、授权令牌和读取的页数，默认值为5。

```
def get_links(subreddit, token, n_pages=5):
```

创建列表，存储新闻报道。

```
    stories = []
```

我们在第7章见过如何在Twitter的API中使用分页功能。Twitter在返回结果时，同时返回一个游标，调用接口时，再一并传递游标，Twitter再利用此游标获取当前结果的下一页。reddit的API游标用法几乎和Twitter一致，只不过它使用`after`参数。获取第一页数据时用不到这个参数，将其置为none，获取到第一页后，再给它传一个有意义的值。代码如下：

```
    after = None
```

遍历我们想得到的那些页。

```
    for page_number in range(n_pages):
```

在上述循环中，设置好头部和URL，方法跟之前一样。

```
        headers = {"Authorization": "bearer
        {}".format(token['access_token']),
            "User-Agent": USER_AGENT}
        url = "https://oauth.reddit.com/r/{}?limit=100".
            format(subreddit)
```

从第二次循环开始，我们需要设置`after`参数（否则返回结果都一样）。下一次循环所用到的`after`的值在前一次循环中设置。如果`after`值不为none，把它添加到URL的末尾，告诉reddit我们接下来要获取哪页的数据。代码如下：

```
        if after:
            url += "&after={}".format(after)
```

下面跟之前一样，使用`requests`库发起请求，在返回结果上调用`json()`方法，得到Python字典。

```
        response = requests.get(url, headers=headers)
        result = response.json()
```

返回结果中包含下一轮迭代所需的`after`值，用它来更新`after`参数。

```
        after = result['data']['after']
```

暂停两秒钟，防止超出API使用频率限制。

```
        sleep(2)
```

在循环体的最后，从reddit的返回结果中获取到每篇报道，把它们添加到`stories`列表中。

我们仅需要标题、URL和喜欢数这三项数据。代码如下：

```
        stories.extend([(story['data']['title'], story['data']['url'],
story['data']['score'])
                        for story in result['data']['children']])
```

最后（循环体外面），返回找到的所有新闻报道。

```
    return stories
```

调用get_links函数时传入栏目名称和授权令牌即可。

```
stories = get_links("worldnews", token)
```

返回结果包含500篇新闻报道的标题、URL等，我们接下来就可以获取到这些URL所指向页面的文本内容。

10.2 从任意网站抽取文本

我们从reddit收集到的网址所指向的网站分属不同的网站组织。这些网站的目标用户是普罗大众而不是计算机程序。当我们尝试用程序获取里面的实际内容时，可能会遇到种种困难，因为如今的网站，有很多逻辑是在后台运行：调用JavaScript库，应用样式表，用AJAX加载广告，在侧边栏增加很多内容等，这些功能增加了网站的复杂程度。这些技术的应用使得当今的Web看起来鲜活、生动、丰富多彩，却增加了自动采集信息的难度。

10.2.1 寻找任意网站网页中的主要内容

我们首先需要访问每个链接，下载各个网页，将它们保存到Data文件夹中事先建好的用于存放原始网页的文件夹raw。后面，我们就要从这些原始网页中获取有用的信息。先把全部网页都保存下来，比起后面时不时地下载网页方便多了。首先，指定存放原始网页的目录。

```
import os
data_folder = os.path.join(os.path.expanduser("~"), "Data",
"websites", "raw")
```

后面我们要用MD5散列算法为每篇报道创建一个唯一的文件名，所以先导入hashlib。散列函数将输入（包含新闻报道名称的字符串）转换为一个看上去像是随机产生的字符串。对于相同的输入，散列函数返回相同的结果，但是不同输入之间的微小差别将会导致截然不同的输出结果。并且，散列函数为单向函数，根据散列值无法得到原来的值。导入hashlib，代码如下：

```
import hashlib
```

对于网页下载失败的网站，我们直接跳过。为了避免错过太多网页，我们维护一个计数器，统计下载失败的次数。如果是系统本身的问题阻止下载，那我们就要解决这些问题。如果失败次

数过多，我们就要找出到底是什么东西在作怪，尝试去解决问题。例如，如果计算机没有联网，所有的500次下载都会失败，你得先解决联网问题，才能再开展下面的工作！

如果所有网页都能成功下载，计数器输出值应该为0。

```
number_errors = 0
```

接下来，遍历每一篇新闻报道。

```
for title, url, score in stories:
```

对报道的标题进行散列操作，作为输出文件名，以保证唯一性。因为reddit网站不要求文章标题具有唯一性，因此两篇报道可能使用相同的标题，这就会导致数据集中两条数据之间的冲突。我们使用MD5算法对报道的URL进行散列操作。现在人们发现MD5也不是绝对可靠的，但是就我们这么小的数据规模来说不太可能出问题（出现碰撞情况），即使出现这样的问题，也不用太担心。

```
    output_filename = hashlib.md5(url.encode()).hexdigest()
    fullpath = os.path.join(data_folder, output_filename + ".txt")
```

接下来下载网页，保存到输出文件夹中。

```
    try:
        response = requests.get(url)
        data = response.text
        with open(fullpath, 'w') as outf:
            outf.write(data)
```

如果下载网页时出现问题，跳过问题网站，继续下载下一个网站的网页。上述代码能够完成我们收集到的95%的网站下载任务，对本章的应用来说足够了，因为我们寻找的是大体趋势而不是做精准研究。有时我们必须下载到所有网站的内容，这时就得调整代码，以处理各种可能的错误。抓取失败的5%～10%的网站，需要编写更为复杂的代码才能进行处理。我们来捕获出现的错误（这可是因特网，会有很多意想不到的错误），增加计数器计数，继续下载下个网站的网页。

```
    except Exception as e:
        number_errors += 1
        print(e)
```

如果错误很多，把上面`print(e)`那一行改为`raise`，调用异常中断机制，以便修改问题。

现在，我们原始网页文件夹raw中有很多网页，查看这些网页（用文本编辑器打开），你会发现新闻报道的内容湮没在HTML、JavaScript、CSS以及其他内容之中。因为我们只对报道本身感兴趣，就需要一种从不同网站的网页中抽取内容的方法。

10.2.2　组装起来

获得原始网页后，我们需要找出每个网页中的新闻报道内容。有一些在线资源使用数据挖掘

方法来解决这个问题，资源列表请见附录。一般来说，很少需要用到这些复杂的算法，当然使用它们能得到更为精确的结果。这也是数据挖掘艺术的一部分——知道什么时候用，什么时候不用。

首先，获取到raw文件夹中的所有文件名。

```
filenames = [os.path.join(data_folder, filename)
             for filename in os.listdir(data_folder)]
```

接着，创建输出文件夹，新闻报道内容抽取出来后，将保存到该文件夹下。

```
text_output_folder = os.path.join(os.path.expanduser("~"), "Data",
                                  "websites", "textonly")
```

接着，编写从网页中抽取文本的代码。我们将使用lxml包解析HTML文件，lxml的HTML解析器容错能力强，可以处理不规范的HTML代码。导入语句如下所示：

```
from lxml import etree
```

文本抽取分为以下三步：首先，遍历HTML文件的每一个节点，抽取其中的文本内容。其次，跳过JavaScript、样式和注释节点，这些节点不太可能包含对我们有价值的信息。最后，确保文本内容长度至少为100个字符。分析文章主题，100个字符就够用，但是要想得到更准确的结果，文章长度有待增加。

前面说过，我们对脚本、样式或注释不感兴趣。因此，创建列表，存放这些不可能包含新闻报道内容的节点。代码如下：

```
skip_node_types = ["script", "head", "style", etree.Comment]
```

我们创建一个函数，把HTML文件解析成lxml etree对象，然后创建另外一个函数，解析前面得到的树，寻找文本。第一个函数简单易懂：打开网页文件，用lxml库的parse方法解析HTML文件。代码如下：

```
def get_text_from_file(filename):
    with open(filename) as inf:
        html_tree = lxml.html.parse(inf)
    return get_text_from_node(html_tree.getroot())
```

上述函数的最后一行，调用getroot()函数，获取到树的根节点，而不是整棵树etree，这样文本抽取函数get_text_from_node以节点作为参数，它能处理包括根节点在内的所有节点，便于递归调用。

文本抽取函数将在任何子节点上调用自己，以抽取子节点中的文本内容，最后返回拼接在一起的所有子节点的文本。

如果一个节点没有任何子节点，文本抽取函数返回该节点的文本内容。如果该节点不包含任何内容，就返回空字符串。请注意，还需要执行上面提到的文本抽取的第三步——检查文本长度

是否达到100个字符。代码如下：

```
def get_text_from_node(node):
    if len(node) == 0:
        # No children, just return text from this item
        if node.text and len(node.text) > 100:
            return node.text
        else:
            return ""
```

明确节点有子节点之后，就对每一个子节点递归调用文本抽取函数，把返回的文本内容拼接在一起。代码如下：

```
    results = (get_text_from_node(child) for child in node
                    if child.tag not in skip_node_types)
    return "\n".join(r for r in results if len(r) > 1)
```

上面`return`语句中的`if`语句是为了防止返回空行（例如，有的节点没有子节点和文本内容）。

遍历所有的原始网页，从文件里抽取文本，把结果保存到纯文本文件[①]夹中。

```
for filename in os.listdir(data_folder):
    text = get_text_from_file(os.path.join(data_folder, filename))
    with open(os.path.join(text_output_folder, filename), 'w')
      as outf:
        outf.write(text)
```

打开纯文本文件夹中的文件，查看它们是否有内容。如果很多都不包含新闻报道内容，提升最少字符限制，我们刚才用的是100。如果设置为更高的值，仍不起作用，或者你的应用对抽取文本内容有更高的要求，请尝试使用附录介绍的更为复杂的方法。

10.3　新闻语料聚类

本章的目的是通过聚类发现新闻语料中潜藏的趋势。我们将使用诞生于1957年的经典机器学习算法k-means（k均值）。

聚类属于无监督学习，我们使用聚类算法探索隐藏在数据里的奥秘。我们的数据集由大约500篇新闻报道组成，人工查看这些文章的主题费时费力。即使使用概括统计方法也不容易，对这种方法而言数据还是相当多。而聚类分析则可以按照主题把它们分成不同的簇，然后就可以按簇研究它们的主题。

我们在没有明确的类别的情况下会使用聚类方法。从这个意义上来讲，聚类算法学习时没有明确的方向性，它们根据目标函数而不是数据潜在的含义来学习。因此，选择聚类效果好的特征

① 指去除掉网页代码后剩下的文本内容。——译者注

就很有必要。对于有监督学习，即使你选用了效果较差的特征，学习算法可以选择不用这些特征。例如，支持向量机为对分类用处不大的特征分配很小的权重。然而，聚类算法会综合所有特征给出最后结果——即使那些不会提供给我们答案的特征。

在对真实的数据进行聚类分析前，最好了解哪些特征适用于当前任务。本章使用词袋模型。我们寻找的是主题相关的簇，因此使用主题相关的特征为数据建模。我们之所以知道这些特征可用来聚类，是因为别人用有监督方法解决类似问题时用过这些特征。对比来说，如果要按照作者把作品分组，我们就要使用类似第9章用到的特征。

10.3.1 k-means 算法

k-means聚类算法迭代寻找最能够代表数据的聚类质心点。算法开始时使用从训练数据中随机选取的几个数据点作为质心点。k-means中的k表示寻找多少个质心点，同时也是算法将会找到的簇的数量。例如，把k设置为3，数据集所有数据将会被分成3个簇。

k-means算法分为两个步骤：为每一个数据点[①]分配簇标签，更新各簇的质心点。

分簇这一步中，我们为数据集的每个个体设置一个标签，把它和最近的质心点联系起来。对于距离质心点1最近的个体，我们为它们分配标签1，距离质心点2最近的个体，分配标签2，以此类推。标签相同的个体属于同一个簇，所有带有标签1的数据点属于簇1（只是暂时的，数据点所属的簇会随着算法的运行发生变化）。

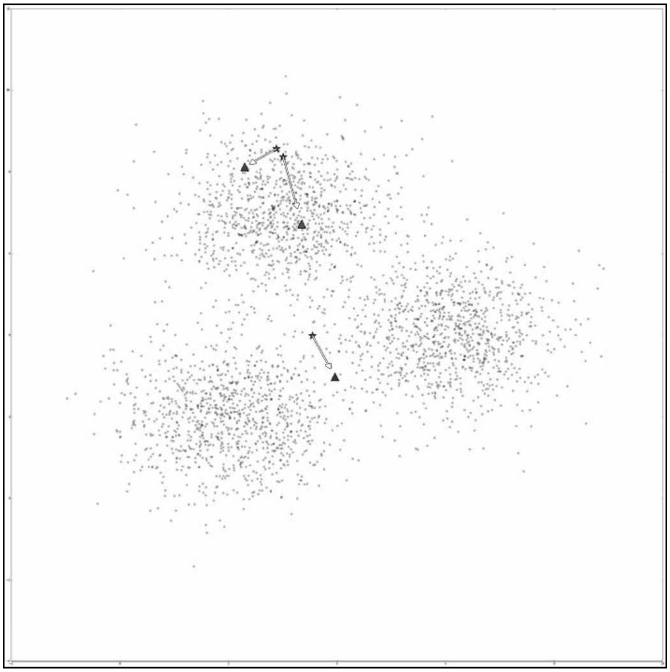
更新环节，计算各簇内所有数据点的均值，更新质心点。

k-means算法会重复上述两个步骤；每次更新质心点时，所有质心点将会小范围移动。这会轻微改变每个数据点在簇内的位置，从而引发下一次迭代时质心点的变动。这个过程会重复执行直到条件不再满足时为止。通常是在迭代一定次数后，或者当质心点的整体移动量很小时，就可以终止算法的运行。有时可以等算法自行终止运行，这表明簇已经相当稳定——数据点所属的簇不再变动，质心点也不再改变时。

下图表示的是对随机创建的数据集执行k-means算法，数据集包含三个簇。图中的三颗五角星表示最初随机从数据集中选取的三个质心点。k-means算法经过5轮迭代后，质心点发生了改变，具体位置用三角形来表示。

---

① 一个数据点对应数据集中一条数据，把数据集看成样本，一条数据即可被称作是一个个体。——译者注



k-means算法因它所使用的数学方法和久经考验而充满魅力。该算法只有（大体上可以这么说）一个参数，虽然距它提出至今已过了半个多世纪，但由于它在很多数据挖掘问题上效果很好，仍被频繁使用。

scikit-learn实现了k-means算法，直接从cluster模块导入即可。

```
from sklearn.cluster import KMeans
```

我们顺便把CountVectorizer类的好兄弟TfidfVectorizer导进来。这个向量化工具根据词语出现在多少篇文档中，对词语计数进行加权。出现在较多文档中的词语权重较低（用文档集数量除以词语出现在的文档的数量，然后取对数）。对于很多文本挖掘应用，使用该种权重计算方法，能够有效提升效果。代码如下：

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

创建数据分析流水线，流水线分两步，第一步是特征抽取，第二步是调用k-means算法。代码如下：

```
from sklearn.pipeline import Pipeline
n_clusters = 10
pipeline = Pipeline([('feature_extraction', TfidfVectorizer(max_
df=0.4)),
                     ('clusterer', KMeans(n_clusters=n_clusters))
                     ])
```

我们为参数max_df设置了一个很低的值0.4，表示忽略出现在40%及以上的文档中的词语。这个参数可用来剔除本身不含有主题相关含义的词语。

> 删除在40%及以上文档中出现的词语将会删除功能词，这种预处理对于第9章将有害无益。

首先训练算法，然后再用它来做预测。前几章的分类任务都是按照这两个步骤实施，但是这里有一点不同——我们没有为fit函数指定目标类别。这也正是无监督学习任务的含义所在！代码如下：

```
pipeline.fit(documents)
labels = pipeline.predict(documents)
```

变量labels包含每个数据点的簇标签。标签相同的数据点属于同一个簇。需要指出的是簇标签本身没有含义：不能说簇1和簇2比簇1和簇3更相似。

我们可以使用Counter类来查看每个簇有多少数据点。

```
from collections import Counter
c = Counter(labels)
for cluster_number in range(n_clusters):
    print("Cluster {} contains {} samples".format(cluster_number,
c[cluster_number]))
```

聚类问题的结果（注意你的数据集可能跟我的不同）往往是一个簇很大，包含了大多数个体，再有几个中等规模的簇，最后还有几个只包含一两个个体的很小的簇，这种不平衡性很常见。

10.3.2　评估结果

聚类分析主要是探索性分析，因此很难有效地评估聚类算法结果的好坏。评估算法结果最直接的方式是根据它要学习的标准对其进行评价。

> 如果你有测试集，你可以对其进行聚类分析来评价效果。更多细节请见http://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html。

对于k-means算法，寻找新质心点的标准是，最小化每个数据点到最近质心点的距离。这叫作算法的惯性权重（inertia），任何经过训练的KMeans实例都有该属性。

```
pipeline.named_steps['clusterer'].inertia_
```

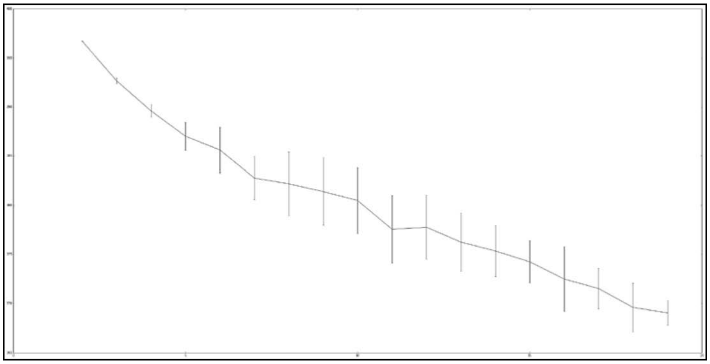
输出结果为343.94，不过这个值本身没有意义，但是可以用它来确定分为多少簇合适。前面的例子，n_clusters的值被设置为10，但这是最佳值吗？下面代码n_clusters依次取2到20之

间的值，每取一个值，k-means算法运行10次。每次运行算法都记录惯性权重。

对于n_clusters变量的每个取值，仅训练X矩阵一次，以（极大）提升代码运行速度。

```
inertia_scores = []
n_cluster_values = list(range(2, 20))
for n_clusters in n_cluster_values:
    cur_inertia_scores = []
    X = TfidfVectorizer(max_df=0.4).fit_transform(documents)
    for i in range(10):
        km = KMeans(n_clusters=n_clusters).fit(X)
        cur_inertia_scores.append(km.inertia_)
    inertia_scores.append(cur_inertia_scores)
```

变量inertia_scores存储了n_clusters取2到20每个值时所对应的惯性权重。我们把惯性权重和簇的数量做成图，以便了解它们之间的关系。



整体而言，随着簇数量的增加，质心点和其他数据点位置的调整逐渐减少，惯性权重应该逐渐降低，这是很容易就能从上图得到的结论。从6簇进一步分为7簇时，质心点是随机选取的，这将会直接影响到最终结果。尽管如此，整体的趋势（你的结果可能会有所不同）是簇数量为6时，惯性权重进行了最后一次大的调整。

随后惯性权重改变很小，虽然没有明确的标准可言。这样的时刻被称作是拐点（elbow），用图来表示就是曲线的顶点，看起来就像是肘部。有些数据集拐点很明显，但是有的数据集可能没有拐点（它们的图像看起来很平滑）。

根据上述分析，把n_clusters的值设置为6，重新运行算法。

```
n_clusters = 6
pipeline = Pipeline([('feature_extraction',
  TfidfVectorizer(max_df=0.4)),
                     ('clusterer', KMeans(n_clusters=n_clusters))
                     ])
pipeline.fit(documents)
labels = pipeline.predict(documents)
```

10.3.3　从簇中抽取主题信息

现在我们把关注点放到各簇上，尝试从中找到每个簇的主题。首先，从特征提取这一步抽取词表。

```
terms = pipeline.named_steps['feature_extraction'].get_feature_names()
```

计算每簇所包含的个体数量。

```
c = Counter(labels)
```

遍历所有的簇，输出每簇所包含的个体数量。评估结果时，要把簇的大小考虑进去——有的簇可能只有一个个体，因此不能代表新闻趋势。代码如下：

```
for cluster_number in range(n_clusters):
    print("Cluster {} contains {} samples".format(cluster_number,
c[cluster_number]))
```

接下来（在循环体内部），遍历该簇最重要的词语。首先，从质心点找出特征值最大的5个特征。代码如下：

```
    print("  Most important terms")
    centroid = pipeline.named_steps['clusterer'].cluster_centers_
[cluster_number]
    most_important = centroid.argsort()
```

依次输出这5个特征。

```
    for i in range(5):
```

对i取反，因为most_important数组中的最小值排在最前面。

```
        term_index = most_important[-(i+1)]
```

然后输出序号、词语和得分。

```
        print("  {0}) {1} (score: {2:.4f})".format(i+1, terms[term_
index], centroid[term_index]))
```

结果能较好地反映当时人们关注的焦点。我得到的结果（2015年3月）中，几个簇的主题为健康问题、中东紧张局势、朝鲜半岛紧张局势和俄罗斯相关问题。这些充斥着当时的新闻头条——虽然很多年以来就一直这样！

10

10.3.4　用聚类算法做转换器

另外提一下，k-means算法（以及其他聚类算法）还有比较有趣的一个特点，就是可以用来简化特征。特征简化的方法有很多种，比如主要成分分析（Principle Component Analysis）、潜在语义索引（Latent Semantic Indexing）等，这些方法还能用来创建新特征，但是它们通常对计算

能力要求很高。

在前面的例子中，词表共有14 000个词条——这个数据集很大。我们的聚类算法把它们仅归为6个簇。我们可以使用数据点到质心点的距离作为特征创建一个特征数较之前少得多的数据集。

在k-means实例上调用转换函数。因为流水线最后一步是`k-means`实例，因此可以在它上面调用转换函数。

```
X = pipeline.transform(documents)
```

上面代码在流水线最后一步的`k-means`实例上调用转换方法。得到的矩阵有六个特征，数据量跟文档的长度相同。

你可以对得到的矩阵进行二次聚类，如果有目标类别的话，也可以用来分类。大致的流程是使用标注好的数据选取特征，用聚类方法把特征数减少到更容易操作的范围内，再用SVM等算法对前面处理过的数据集进行分类。

10.4　聚类融合

第3章研究了如何把几棵效果相对较差的决策树分类器整合在一起形成随机森林，用来处理分类任务。聚类算法也可以进行融合，这样做的主要原因是，融合后得到的算法能够平滑算法多次运行所得到的不同结果。正如我们之前所说，多次运行k-means算法得到的结果因最初选择的质心点不同而不同。多次运行算法，综合考虑所得到的多个结果，可以减少波动。

聚类融合方法还可以降低参数选择对最终结果的影响。大多数聚类算法对参数选择很敏感，参数稍有不同将带来不同的聚类结果。

10.4.1　证据累积

最基本的融合方法是对数据进行多次聚类，每次都记录各个数据点的簇标签。然后计算每两个数据点被分到同一个簇的次数。这就是*证据累积*算法（Evidence Accumulation Clustering，EAC）的精髓。

证据累积算法两大步骤如下：第一步，使用k-means等低水平的聚类算法对数据集进行多次聚类，记录每一次迭代两个数据点出现在同一簇的频率，将结果保存到共协矩阵（coassociation）中。第二步，使用另外一种聚类算法——分级聚类对第一步得到的共协矩阵进行聚类分析。分级聚类一个比较有趣的特性是，它等价于寻找一棵把所有节点连接到一起的树，并把权重低的边去掉。

遍历所有标签，记录具有相同标签的两个数据点的位置，创建共协矩阵。我们需要用到SciPy

的稀疏矩阵csr_matrix。

```
from scipy.sparse import csr_matrix
```

注意函数声明中的参数为所有数据点的标签。

```
def create_coassociation_matrix(labels):
```

记录具有相同标签的两个数据点的行号和列号，把它们分别存储到rows和cols列表中。稀疏矩阵通常由一系列记录非零值位置的列表组成，csr_matrix就是这种类型的。

```
    rows = []
    cols = []
```

标签去重后，再进行遍历。

```
    unique_labels = set(labels)
    for label in unique_labels:
```

查找具有某一标签的所有数据点。

```
        indices = np.where(labels == label)[0]
```

对于每组标签相同的数据点，记录它们的位置。代码如下：

```
        for index1 in indices:
            for index2 in indices:
                rows.append(index1)
                cols.append(index2)
```

在所有循环的外面创建数据集，两个数据点标签相同时，数据集中该位置的值为1。有多少组数据点标签相同，数据集中就有多少个元素为1。代码如下：

```
    data = np.ones((len(rows),))
    return csr_matrix((data, (rows, cols)), dtype='float')
```

调用上述函数，传入所有的数据点标签，就能得到共协矩阵。

```
C = create_coassociation_matrix(labels)
```

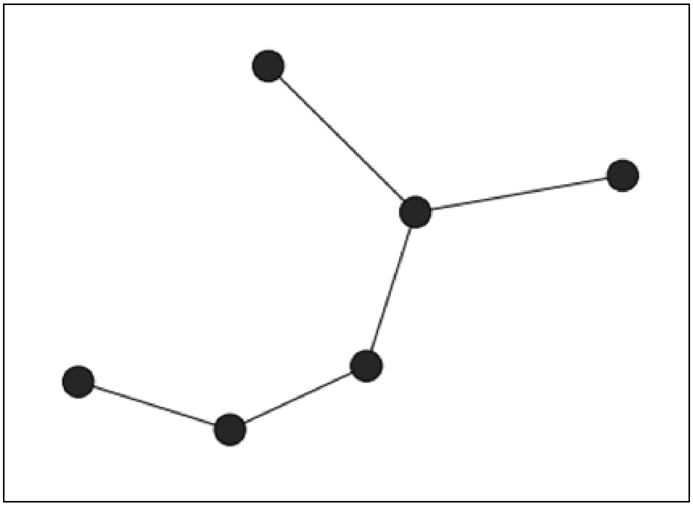
现在我们可以把这些矩阵的多个实例整合起来，多次运行k-means算法，把结果结合起来看。运行C（在记事本新格子里输入C，并运行），查看有多少个非零值。从得到的结果来看，矩阵中大约半数的项有值，我的聚类结果有一个大的簇（各簇的大小越均匀，非零值数量越少）。

下一步对共协矩阵进行分级聚类。我们需要找到该矩阵的最小生成树，删除权重低于阈值的边。

从图的理论角度看，生成树为所有节点都连接到一起的图。最小生成树（Minimum Spanning Tree，MST）即总权重最低的生成树。结合我们的应用来讲，图中的节点对应数据集中的个体，

边的权重对应两个顶点被分到同一簇的次数——也就是共协矩阵所记录的值。

下图的MST有6个节点。图中的节点可以在MST中多次使用，唯一的要求是所有的节点应该连接在一起。



我们使用SciPy sparse包的minimum_spanning_tree函数计算MST。

```
from scipy.sparse.csgraph import minimum_spanning_tree
```

可以直接在coassociation函数返回的稀疏矩阵上调用mst函数。

```
mst = minimum_spanning_tree(C)
```

然而，矩阵C中，值越高表示一组数据点被分到同一簇的次数越多——这个值表示相似度。相反，minimum_spanning_tree函数的输入为距离，高的值反而表示相似度越小。因此，我们对C取反再计算最小生成树。

```
mst = minimum_spanning_tree(-C)
```

上述函数返回结果为一个矩阵，大小跟C相同（行列数分别跟数据集样本数量和特征数相同），只不过保留了最小生成树中的边，其他都被删除了。

然后我们删除其边的权重小于阈值的节点。方法是，遍历MST矩阵中的每一条边，删除低于规定值的边。仅对共协矩阵（值为1或0，没有什么可处理的）进行一次遍历无法完成上述任务。因此，我们来创建额外的标签，创建一个新共协矩阵，然后把这两个矩阵相加。代码如下：

```
pipeline.fit(documents)
labels2 = pipeline.predict(documents)
C2 = create_coassociation_matrix(labels2)
C_sum = (C + C2) / 2
```

然后再计算MST，删除在这两个矩阵中都没有出现的边。

```
mst = minimum_spanning_tree(-C_sum)
mst.data[mst.data > -1] = 0
```

我们需要移除在C1和C2都没有出现的边——也就是值为1的。然而我们刚才在计算时，对C_sum取反，因此，阈值也应该使用相反数。

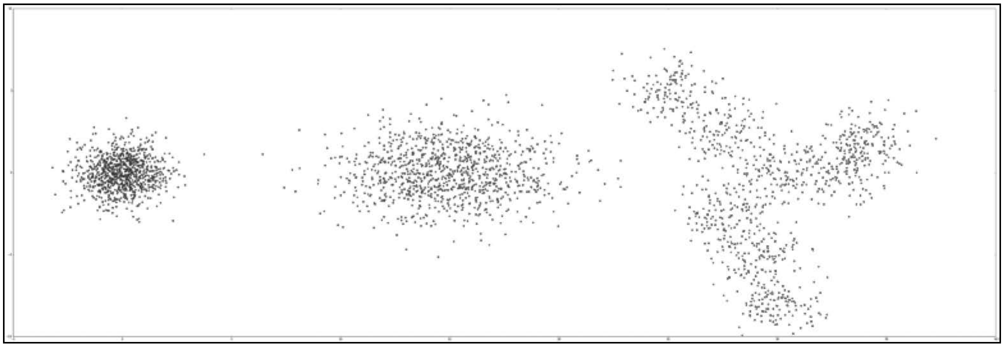
最后，我们找到所有的连通分支，也就是寻找移除低权重的边以后仍然连接在一起的节点。返回的第一个值为连通分支的数量（也就是有多少个簇），第二个值为每个数据点的标签。代码如下：

```
from scipy.sparse.csgraph import connected_components
number_of_clusters, labels = connected_components(mst)
```

我的数据集最终找到了8个簇，每个簇都跟之前差不多。这不足为奇，因为我们仅进行了两轮 k-means迭代，更多的迭代（请见下节）结果差异会更明显。

10.4.2 工作原理

k-means算法不考虑特征的权重，实际上它假定所有的特征取值范围相同。我们在第2章探讨过使用取值范围不同的特征所带来的问题。k-means算法寻找的是圆形簇（circular clusters），如下图所示。



从上图中，我们可以看到不是所有的簇都是圆形。左侧的簇呈圆形，这也是k-means算法很擅长处理的。中间为椭圆形，通过特征规范化（feature scaling），k-means算法可以处理该种聚类问题。最后一幅图甚至不是凸形的——这种形状很奇怪，用k-means算法来处理聚类结果为这种形状的数据集有难度。

证据累积算法的工作原理为重新把特征映射到新空间，上节讲到用k-means来做特征简化，本质上，证据累积算法使用与其相似的原则，每次运行k-means算法都相当于使用转换器对特征进行一次转换。但是我们这里仅使用实际的标签而不是到每个质心点的距离。我们使用存储在共

协矩阵中的数据。

证据累积算法只关心数据点之间的距离而不是它们在原来特征空间的位置。对于没有规范化过的特征，仍然存在问题。因此，特征规范很重要，无论如何都要做（我们用tf-idf规范特征值，从而使特征具有相同的值域）。

我们在第9章见过使用SVM内核做类似的转换。这些转换方法很强大，在处理复杂数据集时记得使用。

10.4.3　实现

现在我们来把前面这些整合起来，创建接口跟scikit-learn对得上的简易聚类算法，实现证据累积的两个步骤。首先，使用scikit-learn的ClusterMixin创建证据累积算法类。

```
from sklearn.base import BaseEstimator, ClusterMixin
class EAC(BaseEstimator, ClusterMixin):
```

参数分别为第一步k-means算法运行次数（创建共协矩阵），用来删除边的阈值和每次运行k-means算法要找到的簇的数量。指定n_clusters的取值范围，每次运行k-means算法都能得到不同的聚类结果。一般而言，从融合的角度看，每次结果有所不同是件好事。否则，融合多次算法跟只运行一次算法效果相同（但是几个聚类结果之间差距较大也不能表明融合后效果就会好）。代码如下：

```
    def __init__(self, n_clusterings=10, cut_threshold=0.5, n_
clusters_range=(3, 10)):
        self.n_clusterings = n_clusterings
        self.cut_threshold = cut_threshold
        self.n_clusters_range = n_clusters_range
```

为EAC类定义fit函数。

```
    def fit(self, X, y=None):
```

接着，使用k-means算法进行低水平聚类，把每一次迭代得到的共协矩阵加起来。为了节省内存，我们使用生成器，仅在需要时创建共协矩阵。生成器每迭代一次，就在数据集上跑一个新的k-means实例，然后创建一个共协矩阵。再使用sum方法把得到的多个共协矩阵加起来。代码如下：

```
        C = sum((create_coassociation_matrix(self._single_
clustering(X))
                  for i in range(self.n_clusterings)))
```

跟前面一样，我们创建MST，删除低于阈值的边（注意使用相反数），找到连通分支。跟scikit-learn其他fit函数一样，我们都需要返回self，以便流水线的下一个步骤使用。代码如下：

```
        mst = minimum_spanning_tree(-C)
        mst.data[mst.data > -self.cut_threshold] = 0
        self.n_components, self.labels_ = connected_components(mst)
        return self
```

接着，编写通过一轮迭代进行聚类的函数。具体方法是，使用numpy的randint函数随机选取几个簇，用参数n_clusters_range指定随机选取的范围。然后使用k-means算法进行聚类和预测数据集属于哪个簇。最后返回由k-means计算得到的簇标签。代码如下：

```
    def _single_clustering(self, X):
        n_clusters = np.random.randint(*self.n_clusters_range)
        km = KMeans(n_clusters=n_clusters)
        return km.fit_predict(X)
```

像之前那样创建流水线，只不过在流水线最后一步，使用证据累积算法而不是之前的k-means算法实例，运行代码。代码如下。

```
pipeline = Pipeline([('feature_extraction', TfidfVectorizer(max_
df=0.4)),
                     ('clusterer', EAC())
                     ])
```

10.5　线上学习

有时，在开始学习之前，我们没有足够的数据用来进行训练。有时，数据量太大，内存装不下，或一时拿不到数据，或完成预测后，我们又拿到了新数据。以上情况就可以使用线上学习方法，在需要时及时训练模型。

10.5.1　线上学习简介

线上学习是指用新数据增量地改进模型。支持线上学习的算法可以先用一条或少量数据进行训练，随着更多新数据的添加，更新模型。相比之下，不支持线上学习的算法在开始训练之前需要一次性拿到所有数据。标准的k-means算法以及前几章的绝大部分算法都不支持线上学习。

线上学习算法在只有几条新数据的情况下就能做到部分更新已有模型。神经网络算法是支持线上学习的标准例子。随着一条新数据输入到神经网络后，网络中的权重根据学习速率进行更新，学习速率通常为一个很小的值，比如0.01。这表明一条新数据只能对模型带来很小的变动（希望是改进）。

神经网络还可以按照批模式来训练，每次只用一组数据进行训练。批模式，算法运行速度快，但是耗内存较多。

同理，我们可以用一个数据点或少量数据点来轻微更新k-means中的质心点。具体做法是，

在k-means算法更新质心点这一步加入学习速率。我们假定这些新数据点是从总体中随机选取的，质心点应该朝它们在原来标准、离线的k-means算法中的位置移动。

线上学习与流式学习（streaming-based learning）有关，但有几个重要的不同点。线上学习能够重新评估先前创建模型时所用到的数据，而对于后者，所有数据都只使用一次。

10.5.2　实现

`scikit-learn`提供了`MiniBatchKMeans`算法，可以用它来实现线上学习功能。这个类实现了`partial_fit`函数，接收一组数据，更新模型。相反，调用`fit()`将会删除之前的训练结果，重新根据新数据进行训练。

因为`MiniBatchKMeans`聚类过程跟`scikit-learn`中其他聚类算法一致，所以创建和使用方法跟其他算法相同。

因此，我们可以使用`TfIDFVectorizer`从数据集中抽取特征，创建矩阵X。代码如下：

```
vec = TfidfVectorizer(max_df=0.4)
X = vec.fit_transform(documents)
```

再来导入`MiniBatchKMeans`，初始化一个实例。

```
from sklearn.cluster import MiniBatchKMeans
mbkm = MiniBatchKMeans(random_state=14, n_clusters=3)
```

接着，随机从X矩阵中选择数据，模拟来自外部的新数据。每次取一些数据，更新模型。

```
batch_size = 10
for iteration in range(int(X.shape[0] / batch_size)):
    start = batch_size * iteration
    end = batch_size * (iteration + 1)
    mbkm.partial_fit(X[start:end])
```

在实例上调用`predict()`方法，获得原始数据集的聚类结果。

```
labels  = mbkm.predict(X)
```

然而目前阶段，我们无法在流水线中使用`TfIDFVectorizer`，因为它不是在线算法。为了解决这个问题，我们使用`HashingVectorizer`类，它巧妙地使用散列算法极大地降低了计算词袋模型所需的内存开销。我们不是记录特征名字，比如文档中的词，而是仅记录这些名字的散列值。这样，在我们查看数据集之前，就能知道所有的特征，因为我们已经拿到了所有特征的散列值。大约有2^{18}个散列值，数据量很大，但是使用稀疏矩阵，存储和计算都很容易，因为矩阵中很多项的值都为`0`。

写作本书时，`sickit-learn`的`Pipeline`类无法用于线上学习。不同的应用有着细微差别，

这也就表明不存在万全之策，也就无法封装一个通用的流水线类。相反，我们可以创建自己的能够支持线上学习的流水线子类。我们这个类继承自scikit-learn的Pipeline类。

```
class PartialFitPipeline(Pipeline):
```

创建partial_fit函数，接收输入矩阵、可选的类别（这里用不到）。

```
    def partial_fit(self, X, y=None):
```

我们之前讲过，流水线由一系列转换步骤组成，前一步的输出为下一步的输入。我们把第一个输入设置为X矩阵，接着，用每一个转换器对输入进行转换。

```
        Xt = X
        for name, transform in self.steps[:-1]:
```

然后，转换已有的数据集，继续迭代，直到达到最后一步（我们这里为聚类算法）。

```
            Xt = transform.transform(Xt)
```

接着，在最后一步调用partial_fit函数，返回结果。

```
        return self.steps[-1][1].partial_fit(Xt, y=y)
```

我们现在就可以创建流水线，在线上学习过程使用MiniBatchKMeans和HashingVectorizer。除了新类PartialFitPipeline和HashingVectorizer外，使用线上学习进行聚类分析，其整个过程跟本章其他部分大同小异，只不过每次使用的数据量少多了。

```
pipeline = PartialFitPipeline([('feature_extraction',
HashingVectorizer()),
                             ('clusterer', MiniBatchKMeans(random_
state=14, n_clusters=3))
                             ])
batch_size = 10
for iteration in range(int(len(documents) / batch_size)):
    start = batch_size * iteration
    end = batch_size * (iteration + 1)
    pipeline.partial_fit(documents[start:end])
labels = pipeline.predict(documents)
```

当然这种方法也有不足之处。比如，我们很难知道对于每个簇来说哪些词最为重要。如果想知道的话，这就要用到另外一种特征抽取工具CountVectorizer，先取到每个词的散列值，然后通过散列值而不是词本身查找词的重要性。这样做有点麻烦，并且抵消了我们使用HashingVectorizer节省下来的内存。并且，我们也无法使用之前用过的max_df参数，因为它需要知道特征是什么并一直统计它们。

此外，我们无法在线上学习中使用tf-idf权重。虽然我们可以估计并使用权重，但是也很麻烦。HashingVectorizer算法非常有用，是散列算法的精彩应用。

10.6　小结

本章研究了一种无监督学习方法——聚类。无监督学习用来探索数据而不是分类或预测。我们从reddit网站采集到的数据没有类别，所以无法对其进行分类。我们使用k-means算法把新闻报道分成组，以找到数据中隐藏的主题和趋势。

从reddit网站采集网址数据后，我们发现它们指向不同的网站，因此，需要研制一种适用范围广的网页内容抽取方法。抽取数据时，我们只寻找大块的文本，而没有使用成熟的机器学习方法。有一些有趣的机器学习方法可以改善文本抽取效果，详见附录部分本章的补充内容。我在附录中，针对每一章都给出了后续学习方向及实验效果改善方法，还给出了参考资料，并介绍了读者可能感兴趣的难度更大的应用。

本章还探讨了一种很直观的融合算法——证据累积算法。融合算法可用来处理结果之间的差异，尤其是你不知道怎么选取好的参数时（参数选取对于聚类来说就特别难）。

最后，我们介绍了线上学习。这是通向包括大数据在内的大型机器学习算法的入口，接下来两章会讲大数据。后两章的实验规模很大，在学习模型的同时，还要求学会管理数据。

下一章将从无监督学习再度回到分类。我们将研究以复杂神经网络为基础的分类方法——深度学习。

第 11 章

用深度学习方法为图像中的物体进行分类

第8章用到了最基础的神经网络。最近几年，该领域兴起的研究热潮为其带来了一系列长足的进步。如今，神经网络的研究创造出了一些最为先进和精确的分类算法，在很多领域展露锋芒。

计算机性能的提升使得训练更大、更复杂的神经网络成为可能，但该领域之所以能够取得进步，主要不是因为计算性能的提升，而是因为采用了新的算法和神经网络层。

本章研究如何确定图像中的物体，我们使用像素值作为神经网络的输入值，自动找到有用的像素组合，形成更高层级的特征，然后将其用于实际的分类。本章主要内容如下：

- 为图像中物体分类
- 不同类型的深度神经网络
- `Theano`、`Lasagne`和`Nolearn`：用于创建和训练神经网络的库
- 用GPU提升算法速度

11.1 物体分类

计算机视觉逐渐成为科技领域的明日之星。未来五年，我们也许就能坐上自动驾驶汽车（甚至可能比这还要提前，如果流传的消息是可信的）。要让计算机来开车，它得能看清周围的环境：障碍物、其他车辆和天气状况等。

虽然计算机借助雷达等技术很容易就能检测到是否有障碍物，但这还不够，它还需要知道障碍物是什么。假如是动物，它们可能很快就会跑开，但如果是建筑物，那就麻烦了，计算机需要控制车辆躲开。

11.2 应用场景和目标

本章将构建一个系统，它接收图像，给出图像里面是什么物体。该系统就好比是自动驾驶汽

车的视觉系统，它能发现道路及两侧的任何障碍物。我们使用如下形式的图像。



图像数据来自于著名的CIFAR-10数据集，它含有6万张32像素见方的图像，每个像素都有一个红-绿-蓝（RGB）值。这个数据集已经分为训练集和测试集两部分，我们在训练结束后才会用到测试集，这里先提一下。

> CIFAR-10数据集的下载地址为http://www.cs.toronto.edu/~kriz/cifar。请下载Python版本，所有图像已经转换为`numpy`数组。

我们来看下这些图像数据长什么样，打开一个IPython Notebook笔记本文件，设置好文件名。我们开始只用第一批文件，到后面再增加数量使用全部数据集。

```
import os
data_folder = os.path.join(os.path.expanduser("~"), "Data", "cifar-10-
batches-py")
batch1_filename = os.path.join(data_folder, "data_batch_1")
```

接着，创建一个函数，读取第一批图像文件数据。这些图像数据文件格式为pickle，pickle是Python用来保存对象的一个库。通常在pickle文件上调用`pickle.load`方法就能取得保存在里面的数据。但这里有个小问题：这些pickle文件是Python 2生成的，而我们要用Python3打开。所以在打开时需要把编码设置成latin（虽然我们是以字节模式打开）。

```
import pickle
# Bigfix thanks to: http://stackoverflow.com/questions/11305790/
pickle-incompatability-of-numpy-arrays-between-python-2-and-3
def unpickle(filename):
    with open(filename, 'rb') as fo:
        return pickle.load(fo, encoding='latin1')
```

使用上述函数加载数据集。

```
batch1 = unpickle(batch1_filename)
```

这一批图像文件读进来后为一个字典结构，包含numpy数组形式的图像数据、图像类别、文件名以及该批文件的简短说明（例如，training batch 1 of 5表示训练集共五批，这是第一批）。

以data作为键，从字典batch1中取到存储这一批图像数据的列表后，再用图像索引号就能取到其中一张图像的数据。

```
image_index = 100
image = batch1['data'][image_index]
```

每一张图像数据为一个numpy数组，数组共有3072项，每一项为0到255之间的一个值。每个值代表图像某一位置的红、绿或蓝色的颜色强度。

图像数据格式与matplotlib（绘制图像）所使用的有所不同，因此，用它来显示图像前，需要改变数组的形状，对矩阵进行转换。这种数据格式不会影响神经网络训练（我们定义网络时会考虑支持这种数据格式），但是要用matplotlib绘制图像，就需要对数据格式进行转换。

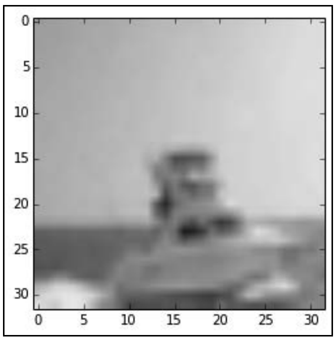
```
image = image.reshape((32,32, 3), order='F')
import numpy as np
image = np.rot90(image, -1)
```

然后，就可以用matplotlib绘制图像。

```
%matplotlib inline

from matplotlib import pyplot as plt
plt.imshow(image)
```

这是张轮船图像。



11

图像分辨率很低——只有32像素见方。尽管如此，大部分人还是能看出图中是一艘船。我们能让计算机辨认出来吗？

你可以修改图像索引查看其他图像，了解下数据集的特点。

本章项目的目标是创建分类系统，预测像上面这样的图像的类别。

使用场景

计算机视觉技术应用范围很广。

在线地图网站，比如谷歌地图就使用计算机视觉技术，自动为街景中的人脸添加模糊效果，以保护被拍摄到的人们的隐私。除此之外，还有别的用途。

人脸识别技术在很多行业都有所应用。如今，照相机都能自动识别人脸，以提升拍摄质量（拍照时用户往往把相机聚焦到脸部）。人脸识别技术还可以用来识别照片中的人脸。Facebook就使用了这项技术，方便用户标记朋友。

我们前面也说过，自动驾驶汽车高度依赖计算机视觉来识别道路，躲避障碍物。计算机视觉是很多行业、应用都在努力解决的一个问题，不只是为了研制自动驾驶汽车，也不只是为了满足消费者的需求，采矿业等行业也在寻求攻克计算机视觉技术难关之路。

除此之外，其他行业也在使用计算机视觉技术，比如仓储业自动检测不合格产品就用到该项技术。

航天工业使用计算机视觉技术实现数据采集的自动化。这对于有效使用航天器来说至关重要，因为从地球上发送信号到火星上的探测车要花费很长时间，有时还会无法送达（例如，地球和火星不是面对面时）。人们跟太空探测器打交道越来越多，而从遥远的地球控制它们很不方便，增加这些设备的自主性就显得很有必要。

下图为美国国家航空和航天局（NASA）设计的火星车，它大量应用了计算机视觉技术。



11.3 深度神经网络

从理论角度看，第8章所使用的神经网络有几个显著的特点。例如，只需要一层隐含层来学习任何类型的映射（虽然隐含层可能规模很大）。神经网络在20世纪七八十年代很火，但随后人们不再使用它们，尤其是跟支持向量机等其他分类算法相比，神经网络被冷落了。首要问题是，运行多个神经网络比其他算法对计算能力有更高的要求，超出了当时计算机的能力范围。

其次是训练神经网络所需工作量很大。虽然那时反向传播算法已经研制出来，处理大型神经网络仍存在问题，需要大量的训练才能确定权重。

这两个问题在近些年都得到了解决，神经网络迎来了又一个春天。计算能力比起30年前更容易获得，训练算法的改进使得在现有计算能力的情况下，能够顺利完成大型神经网络的训练。

11.3.1 直观感受

深度神经网络和第8章中的基本神经网络的差别在于规模大小。至少包含两层隐含层的神经网络被称为深度神经网络。实际工作中遇到的深度神经网络通常规模很大，每层神经元数量和层次都非常多。2000年年中的一些研究，关注的是层次数量多的深度神经网络，而更巧妙的算法能够减少实际所需要的层数。

神经网络接收很基础的特征作为输入——就计算机视觉而言，输入为简单的像素值。 神经网络算法把这些数据整合起来向网络中传输，在这个过程中，基本的特征组合成复杂的特征。有时，这些组合特征对我们来说没有什么含义，但是它们表示个体某些方面的特征，计算机依靠它们进行分类。

11.3.2 实现

由于深度神经网络规模很大，实现起来很有挑战。实现不好，运行时间会很长，甚至还会出现由于内存不足，最后无法运行的情况。

神经网络的基本实现可以从创建神经元类开始，多个神经元组成层类，每一层的神经元使用边这个类的一个实例连接到另一层神经元。这种基于类的实现，有助于显示网络的工作方式，但是对于创建大型网络，效率较低。

神经网络的核心其实就是一系列矩阵运算。两个网络之间连接的权重可以用矩阵来表示，其中行表示第一层的神经元，列表示第二层神经元（有时会用到该矩阵的转置矩阵）。矩阵的每一个元素表示位于不同层的两个神经元之间连接的权重。一个神经网络就可以用一组这样的矩阵来表示。除了神经元外，每层增加一个偏置项，它是一个特殊的神经元，永远处于激活状态，并且跟下一层的每一个神经元都有连接。

对神经网络有了上述这般深入理解，我们就可以使用数学运算创建、训练和使用神经网络，比起前面用类来实现，效率要更高。就矩阵运算而言，有很多非常好用的库，其代码经过充分优化，借助它们可以高效完成矩阵运算。

第8章使用的PyBrain还实现了神经网络的卷积层。但是我们这里要用到的一些功能，它没有提供。对于规模更大，定制化程度更高的神经网络，我们需要更强大的库。因此，我们选用Lasagne和nolearn两个库。这两个库依赖于擅长处理数学表达式的Theano库。

本章首先通过用Lasagne实现一个基础的神经网络，来介绍相关概念。然后，使用nolearn库重新做第8章中的字母识别实验。最后，使用更为复杂的卷积神经网络对CIFAR数据集进行图像分类，为了提升性能我们使用GPU而不是CPU运行算法。

11.3.3 Theano 简介

Theano是用来创建和运行数学表达式的工具。它用起来乍一看跟编写程序没有差别，但是在Theano中，我们定义函数要做什么而不是怎么做，这样Theano就能以最佳的方式对表达式进行求值，还可以进行延迟计算——只在需要时，对表达式求值，而不是定义时。

多数程序员不会每天都使用这种类型的编程范式，但是他们几乎天天接触使用这种编程范式的系统。关系型数据库，尤其是SQL类，就用到了叫作声明型范式（declarative paradigm）的概念。比如程序员定义了含有WHERE子句的SELECT查询语句。数据库解释查询语句，并根据一系列因素创建优化过的查询语句，数据库要考虑的因素有WHERE子句是否建立在主键之上、数据存储格式等。程序员决定他们要什么，系统决定怎么做。

> 你可以使用pip安装Theano：pip3 install Theano。

我们可以用Theano来定义函数，处理标量（scalars）、数组和矩阵及其他数学表达式。例如，我们可以创建计算直角三角形斜边长度的函数。

```
import theano
from theano import tensor as T
```

首先，定义两个输入a和b，它们为简单的数值类型，因此使用标量。

```
a = T.dscalar()
b = T.dscalar()
```

接着，定义输出c，它为由a和b构成的一个表达式。

```
c = T.sqrt(a ** 2 + b ** 2)
```

注意，上述c既不是函数，也不是一个数值——它是由a和b组成的表达式。a和b不是一个确切的值——这是一个代数式，最终结果不确定。为了计算这个表达式，我们来定义一个函数。

```
f = theano.function([a,b], c)
```

上述语句告诉Theano创建一个函数，这个函数接收a、b，返回经过计算得到的结果，也就是输出值c。例如，f(3, 4)，返回5。

虽然上述例子看起来可能不比Python普通函数强大多少，但是我们现在就在代码其他部分和后面的映射关系中使用我们的函数或数学表达式c。此外，虽然我们在创建函数之前，就定义了表达式，其实直到调用函数时，才会对表达式进行计算。

11.3.4 Lasagne 简介

Theano库不是用来创建神经网络的，就好比numpy库不是用来执行机器学习任务的，而是被别的库使用，来出卖苦力的。Lasagne库是专门用来构建神经网络的，它使用Theano进行计算。

Lasagne实现了几种比较新的神经网络层和组成这些层的模块，具体介绍如下。

- 内置网络层（Network-in-network layers）：这些小神经网络比传统的神经网络层更容易解释。
- 删除层（Dropout layers）：训练过程随机删除神经元，防止产生神经网络常见的过拟合问题。
- 噪音层（Noise layers）：为神经元引入噪音，也是为了解决过拟合问题。

本章使用卷积层（convolution layers，层级结构模拟人类视觉工作原理）。卷积层使用少量相互连接的神经元，分析一部分输入值（比如我们这里的一张图像），便于神经网络实现对数据的标准转换，比如对图像数据的转换。视觉分析实验，就是用卷积层对图像进行转换[①]。

比较而言，传统的神经网络内部连接十分紧密——一层的所有神经元全都连接到下一层所有神经元。

lasagne.layers.Conv1DLayer和lasagne.layers.Conv2DLayer classes两个类实现了卷积网络。

> 写作本书时，Lasagne还没有正式版，资源没有上传到pip站点，因此无法用pip安装。但是可以从github安装。把Lasagne代码仓库克隆到本地新建的文件夹中：**git clone https://github.com/Lasagne/Lasagne.git**。进入到Lasagne文件夹，使用下面命令安装：
>
> ```
> sudo python3 setup.py install
> ```
>
> 安装指南请见http://lasagne.readthedocs.org/en/latest/user/installation.html。

① 原文为translating the image。——译者注

神经网络使用卷积层（一般而言，仅卷积神经网络包含该层）和池化层（pooling layer），池化层接收某个区域最大输出值，可以降低图像中的微小变动带来的噪音，减少（down-sample，降采样）信息量，这样后续各层所需工作量也会相应减少。

Lasagne还实现了池化层——比如lasagne.layers.MaxPool2DLayer类。再加上前面的卷积层，现在我们准备好了创建卷积神经网络所需的全部部件。

在Lasagne中创建神经网络比起只用Theano更加容易。我们通过实现一个简单的卷积神经网络，介绍相关规则。我们再次使用第1章所用到的Iris数据集，该数据集很适合用来测试新算法，即使是复杂的深度神经网络也没问题。

首先，打开一个新的笔记本文件。前面加载CIFAR数据集所使用的笔记本文件，先搁一边，后面还会用。

加载Iris数据集。

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data.astype(np.float32)
y_true = iris.target.astype(np.int32)
```

Lasagne对数据类型有特殊要求，因此，需要把类别值转换为int32类型（在原始数据集中用int64类型存储）。

把数据集分为训练集和测试集两部分。

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y_true, random_
state=14)
```

接着，分别创建卷积神经网络各层。我们的数据集有四个特征、三个类别，这样第一层和最后一层神经元数量就确定了，但是中间层多大呢？中间层大小不同，最终结果也会不同，尝试使用不同的值，看看结果会有怎样的变化。

首先，创建输入层，其神经元数量跟数据集特征数量相同。可以指定每一批输入的数量（设置为10），这样Lasagne可以在训练阶段做些优化。

```
import lasagne
input_layer = lasagne.layers.InputLayer(shape=(10, X.shape[1]))
```

接着，创建隐含层。该层从输入层接收输入（由第一个参数指定），该层共有12个神经元，使用非线性的sigmoid函数，我们在第8章曾经介绍过。

```
hidden_layer = lasagne.layers.DenseLayer(input_layer, num_units=12,
nonlinearity=lasagne.nonlinearities.sigmoid)
```

接下来，创建输出层，它接收来自隐含层的输入，输出层共有三个神经元（跟类别的数量一致），使用非线性的softmax函数，该函数主要用于神经网络的最后一层。

```
output_layer = lasagne.layers.DenseLayer(hidden_layer, num_units=3,
                                    nonlinearity=lasagne.
nonlinearities.softmax)
```

依照Lasagne的习惯用法，输出层为我们的神经网络。当我们输入一条数据到神经网络时，它查看输出层，向上回溯找到向输出层提供输入的那一层（第一个参数）。这个过程重复进行直到到达输入层，因为输入层没有上一层，所以就把要处理的数据交给输入层处理。输入层的激活函数把接收到的数据处理后输出给调用它的层（我们这里是隐含层），然后再一步步在网络中传播直到输出层。

为了训练刚创建的网络，我们需要定义几个Theano训练函数。在这之前，需要定义一个Theano表达式和函数。我们先来为神经网络的输入数据、输出结果和实际输出结果声明变量。

```
import theano.tensor as T
net_input = T.matrix('net_input')
net_output = output_layer.get_output(net_input)
true_output = T.ivector('true_output')
```

接着，定义损失函数，训练函数如何提升网络效果需要参考它的返回值——训练神经网络时，以最小化损失函数的返回值为前提。我们用类别交叉熵（categorical cross entropy）表示损失，这是一种衡量分类数据（categorical data）分类效果好坏的标准。损失函数表示的是网络的期望输出和实际输出两者之间的差距。

```
loss = T.mean(T.nnet.categorical_crossentropy(net_output,
  true_output))
```

接着，定义修改网络权重的函数。我们需要获取到网络的所有参数，创建调整权重的函数（使用Lasagne提供的工具），使得损失降到最小。

```
all_params = lasagne.layers.get_all_params(output_layer)
updates = lasagne.updates.sgd(loss, all_params, learning_rate=0.1)
```

最后，创建两个Theano函数，先是训练网络，然后获取网络的输出，以用于后续测试。

```
import theano
train = theano.function([net_input, true_output], loss,
updates=updates)
get_output = theano.function([net_input], net_output)
```

然后调用训练函数，在训练集上进行一轮迭代，接收训练数据，预测类别，与给定类别作比较，更新特征权重，以最小化损失。然后再进行1000次迭代，逐渐改进神经网络。


```
for n in range(1000):
    train(X_train, y_train)
```

接着，对输出计算F值，以评估分类效果。首先获取到所有输出。

```
y_output = get_output(X_test)
```

> 注意，get_output为Theano函数，是我们从神经网络中获取到的，因此，上述代码不用再把神经网络作为参数传进去。

y_output为输出层每个神经元的激励作用（activation）大小。找出激励作用最高的神经元，就能得到预测结果。

```
import numpy as np
y_pred = np.argmax(y_output, axis=1)
```

数组y_pred为预测结果数组，跟我们在前面分类任务中用到的相同。然后，我们就可以计算F1值。

```
from sklearn.metrics import f1_score
print(f1_score(y_test, y_pred))
```

结果异常完美——1.0！这表明我们的算法为测试集中所有数据正确分类，这个结果太棒了（虽然数据集有点小）。

从上面我们可以看到，仅用Lasagne库也能创建、训练一个神经网络，但是有点麻烦。我们可以使用nolearn库，对上述过程做进一步封装，使它很容易就能与scikit-learn接口对得上。

11.3.5　用 nolearn 实现神经网络

nolearn对Lasagne进行了封装，虽然比起用Lasagne创建神经网络，使用nolearn无法对一些细节进行调整，但是代码可读性更强，也更容易管理。

nolearn实现了几种常见的复杂程度很高的神经网络，应该能够满足你的需求。如果你想拥有更多的控制权，可以回头继续使用Lasagne，但是创建和训练过程需要你多花点心思。

为了快速了解nolearn，我们再次来实现第8章中识别图像中字母的实验。我们重建在第8章所创建的密集神经网络（dense neural network）。再次在笔记本文件输入创建数据集的代码。这些代码的具体含义，请参考第8章。

```
import numpy as np
from PIL import Image, ImageDraw, ImageFont
from skimage.transform import resize
from skimage import transform as tf
from skimage.measure import label, regionprops
from sklearn.utils import check_random_state
from sklearn.preprocessing import OneHotEncoder
from sklearn.cross_validation import train_test_split
```

```
def create_captcha(text, shear=0, size=(100, 24)):
    im = Image.new("L", size, "black")
    draw = ImageDraw.Draw(im)
    font = ImageFont.truetype(r"Coval.otf", 22)
    draw.text((2, 2), text, fill=1, font=font)
    image = np.array(im)
    affine_tf = tf.AffineTransform(shear=shear)
    image = tf.warp(image, affine_tf)
    return image / image.max()

def segment_image(image):
    labeled_image = label(image > 0)
    subimages = []
    for region in regionprops(labeled_image):
        start_x, start_y, end_x, end_y = region.bbox
        subimages.append(image[start_x:end_x,start_y:end_y])
    if len(subimages) == 0:
        return [image,]
    return subimages

random_state = check_random_state(14)
letters = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
shear_values = np.arange(0, 0.5, 0.05)

def generate_sample(random_state=None):
    random_state = check_random_state(random_state)
    letter = random_state.choice(letters)
    shear = random_state.choice(shear_values)
    return create_captcha(letter, shear=shear, size=(20, 20)),
letters.index(letter)
dataset, targets = zip(*(generate_sample(random_state) for i in
range(3000)))
dataset = np.array(dataset, dtype='float')
targets =  np.array(targets)

onehot = OneHotEncoder()
y = onehot.fit_transform(targets.reshape(targets.shape[0],1))
y = y.todense().astype(np.float32)

dataset = np.array([resize(segment_image(sample)[0], (20, 20)) for
sample in dataset])
X = dataset.reshape((dataset.shape[0], dataset.shape[1] * dataset.
shape[2]))
X = X / X.max()
X = X.astype(np.float32)

X_train, X_test, y_train, y_test = \
    train_test_split(X, y, train_size=0.9, random_state=14)
```

11

神经网络由一系列的层组成。在nolearn中实现神经网络，只需定义它由Lasagne哪几种类型的层组成就行，跟PyBrain做法大同小异。第8章所创建的神经网络使用的是全连通密集层。nolearn有相应的实现，这表明我们可以用nolearn实现的各层来复制第8章神经网络的基本结

构。首先，创建由输入层、密集隐含层和密集输出层组成的层级结构。

```
from lasagne import layers
layers=[
        ('input', layers.InputLayer),
        ('hidden', layers.DenseLayer),
        ('output', layers.DenseLayer),
        ]
```

导入以下所需模块，用到时再解释它们的作用。

```
from lasagne import updates
from nolearn.lasagne import NeuralNet
from lasagne.nonlinearities import sigmoid, softmax
```

接着，定义神经网络，它与scikit-learn估计器兼容。

```
net1 = NeuralNet(layers=layers,
```

注意上面代码没加右半边括号——我们有意为之。输入神经网络的参数，先从每层的大小开始吧。

```
    input_shape=X.shape,
    hidden_num_units=100,
    output_num_units=26,
```

上述参数跟每一层相对应。换句话说，input_shape参数就会去查找名称为input的输入层，这跟在流水线中设置参数的工作原理很相似。

接着，定义非线性函数。跟之前一样，隐含层使用sigmoid函数，输出层使用softmax函数。

```
    hidden_nonlinearity=sigmoid,
    output_nonlinearity=softmax,
```

接下来，指定偏置神经元，它们位于隐含层，一直处于激活状态。偏置神经元对于网络的训练很重要，神经元激活后，可以对问题做更有针对性的训练，以消除训练中的偏差。举个简单的例子，如果预测结果总是偏差4，我们可以使用–4偏置值抵消偏差。我们设置的偏置神经元就能起到这样的作用，训练得到的权重决定了偏置值的大小。

偏置项为一组权重值，它所包含的元素数量应与它所连接的层的大小相同。

```
    hidden_b=np.zeros((100,), dtype=np.float32),
```

接下来，定义神经网络的训练方式。我们在第8章中用到的训练方法，nolearn没有与之完全相同的，因为nolearn没有削减权重的方法。然而，它的确有冲量（momentum），我们这里使用一个高学习速率和低冲量值。

```
    update=updates.momentum,
    update_learning_rate=0.9,
    update_momentum=0.1,
```

接下来，我们把分类问题定义为回归问题。这看起来有点奇怪，因为这是分类任务。然而，别忘了输出结果是一组数值，在训练阶段把它当成回归问题进行优化比起分类问题效果更好。

```
    regression=True,
```

最后，最大训练步数（epoch）设置为1000，这样既能保证训练效果，又不至于花太长时间（对于我们数据集很合适，但其他数据集训练步数会有所差异）。

```
    max_epochs=1000,
```

终于可以把创建神经网络这个函数的右半边括号加上了。

```
)
```

接着，在训练集上训练网络。

```
net1.fit(X_train, y_train)
```

现在，我们来评估训练得到的网络。首先，获取神经网络的输出，跟Iris分类一样，我们需要使用argmax方法找到输出值中最大的一项，然后找到它所对应的类别。

```
y_pred = net1.predict(X_test)
y_pred = y_pred.argmax(axis=1)
assert len(y_pred) == len(X_test)
if len(y_test.shape) > 1:
    y_test = y_test.argmax(axis=1)
print(f1_score(y_test, y_pred))
```

结果同样振奋人心——在我的计算机上，再次全部预测正确。然而，你可能会得到不同的结果，因为nolearn库具有一定的随机性，眼下无法直接控制。

11.4 GPU 优化

神经网络体积可能会增长得很大，这意味着需要更多内存；然而，使用稀疏矩阵这样高效的存储方式把整个神经网络装进内存一般不会遇到问题。

神经网络体积增长带来的主要问题是计算时间增加。此外，对于有些数据集，神经网络需要更多的训练步数才能较好地拟合数据集。本章所创建的神经网络在我性能还不错的计算机上，每完成一步训练需要八分多钟，而我们需要运行数十步甚至成百上千步训练。体积更大的神经网络，每一步训练需要几个小时，而为了取得最佳效果，可能需要成千上万步训练。

多长时间才能完成训练啊！

好在神经网络最核心的计算类型主要为浮点型运算。此外，由于神经网络训练过程主要为矩阵操作，大量运算都可以并行处理。这些因素表明用GPU进行计算，能提升训练速度。

11.4.1　什么时候使用 GPU 进行计算

GPU的设计初衷是用于图像渲染。这些图像用矩阵及由矩阵组成的数学方程式来表示，它们被转换为我们在屏幕上看到的像素。这个过程涉及大量并行计算。如今的CPU很可能是多核（你的计算机可能有2、4，甚至16个核——或更多！），GPU却拥有专门用于图像处理的成千上万个小核。

CPU每个核单独工作速度往往更快，访问计算机内存等操作效率更高，因而适合处理序列化任务。说实话，让CPU出苦力更容易，因为几乎所有的机器学习库默认使用CPU，如果要用GPU做并行计算，需要很多额外工作，但是它所带来的好处也是显而易见的。

有些任务，包含大量数字运算，但计算量都不大，可以同时处理，这样的任务最好交由GPU来处理。很多机器学习任务都具有类似的特点，因此使用GPU处理能提升算法运行速度。

让代码在GPU上跑起来可不简单，中间可能会遇到各种各样的挫折，具体与GPU类型、配置方式、操作系统相关，有时你可能还需要修改计算机底层设置。

使用GPU运算，主要方法有以下三种。

- 第一种，使用现有计算机。了解你的计算机型号，查找GPU、操作系统相关的软件和驱动程序，寻找相关教程，软件、教程是否能用与操作系统相关。不过，现在使用GPU，比前几年容易多了，有更多更好的软件和驱动程序来启动GPU计算。
- 第二种，选定计算机系统，找到关于如何设置系统的文档，购买一台装有这种系统的新计算机，买的系统很好用，但是可能相当昂贵——如今的计算机，GPU是价格最高的部件之一。尤其是你对性能有更高要求的话——你需要一个价格不菲的好GPU。
- 第三种，使用专门为GPU计算而配置的虚拟机。例如，Markus Beissinger在亚马逊的AWS上搭建了一个这样的系统，该系统不是免费的，你得付费才能使用它提供的服务，但是比起购买一台新计算机要便宜得多。具体费用由你所在的区域、使用的系统类型和资源消耗数量来决定，你可能每小时花不到1美元，往往比这还要少得多。如果你使用AWS的竞价型实例（spot instance），每小时只需几美分（然而你需要单独开发在竞价型实例上运行的代码）。

如果你无法承担虚拟机开销，我建议你使用上面提到的第一种方法，即使用现有的计算机。你还可以试试看能不能从隔段时间就要换装备的家人或朋友那里找到二手GPU（爱玩游戏的朋友很可能有哦！）。

11.4.2　用 GPU 运行代码

我们这里将使用上述第三种方法，在Markus Beissinger系统的基础上创建一个虚拟机，该虚拟机运行在亚马逊的EC2云平台上。除了EC2，还有很多其他Web服务可供使用，但方法可能跟

这里有所不同。我会讲下如何在亚马逊云平台上搭建虚拟机。

如果你打算使用自己的计算机，并且已经配置好，可以用GPU进行计算了，请跳过该节。

> 关于如何进行设置的更多信息，请见http://markus.com/install-theano-on-aws/，也许还提供在其他计算机上启用GPU计算的方法。

下面，看下如何在AWS上创建虚拟机。首先，访问AWS的登录界面。

https://console.aws.amazon.com/console/home?region=us-east-1

用你的AWS账号登录。如果没有，得先创建一个才能继续下面的操作。

接着，访问EC2云平台的控制台：https://console.aws.amazon.com/ec2/v2/home?region=us-east-1。

点击Launch Instance，从右上方的下拉菜单中选择N. California作为目的地。

点击Community AMIs，搜索ami-b141a2f5，这就是Markus Beissinger创建的系统。然后，点击Select。下一屏中的机器类型选择g2.2xlarge，点击Review and Launch。在下一屏，点击Launch。

这个时候，AWS开始向你收费，所以使用完毕后请关机。从EC2云平台选择你刚创建的云主机，先停止运行。机器处于停机状态，不需要支付费用。

系统会给出如何连接虚拟机的相关信息。如果你之前没有使用过AWS，你可能需要创建密钥以安全连接虚拟机。创建时，需要指定密钥的名称，下载.pem格式的密钥文件，将其保存到一个安全地方——一旦丢失，你将无法登录自己的虚拟机！

点击Connect，了解如何使用密钥文件连接虚拟机。你很可能是用如下`ssh`命令登录。

```
ssh -i <certificante_name>.pem ubuntu@<server_ip_address>
```

11.5　环境搭建

连接到虚拟主机后，你可以安装最新的`Lasagne`和`nolearn`库。

首先，使用`git`命令克隆`Lasagne`代码仓库，本章前面用过。

```
git clone https://github.com/Lasagne/Lasagne.git
```

在虚拟机上编译这个库，需要用到Python 3的`setuptools`，我们可以用Ubuntu常用的应用和包管理命令`apt-get`来安装；我们还需要`numpy`的开发版。

在虚拟机的命令行运行下述命令。

```
sudo apt-get install python3-pip python3-numpy-dev
```

接着，安装Lasagne。首先，切换到源代码所在的目录，然后运行setup.py完成安装。

```
cd Lasagne
sudo python3 setup.py install
```

> 我们已经安装了Lasagne，还需要安装nolearn，为了简单起见，把它俩都作为系统包来安装。考虑到可移植性，建议你使用virtualenv安装这些包，这样你可以在同一台虚拟机上使用不同版本的Python和第三方包，把代码移植到其他机器上也更容易。更多信息请见http://docs.python-guide.org/en/latest/dev/virtualenvs/。

Lasagne编译好后，我们就可以安装nolearn。切换到你自己的主目录，照下面的步骤来，跟前面安装Lasagne的步骤相同，只不过要注意这次克隆的是nolearn。

```
cd ~/
git clone https://github.com/dnouri/nolearn.git
cd nolearn
sudo python3 setup.py install
```

我们的环境差不多搭建好了。还需要安装scikit-learn、scikit-image库，这两个库都可以用pip3安装，这些库所依赖的scipy和matplotlib也没有安装。建议使用apt-get来安装scipy和matplotlib，因为使用pip3可能会遇到比较棘手的问题。

```
sudo apt-get install python3-scipy python3-matplotlib
sudo pip3 install scikit-learn scikit-image
```

接着，需要把代码搬到虚拟机上，方法有很多，最简单的莫过于复制粘贴。

首先，打开前面用过的笔记本文件（在你本地的计算机上，不是在亚马逊的虚拟机上）。笔记本文件顶部有一个菜单。点击File，然后再点击Download as，选择Python，将其保存到本地计算机上。该步骤将笔记本文件保存为可以从命令行运行的.py文件。

打开.py文件（有些操作系统，你可能需要右键单击该文件，从弹出的菜单中选择用文本编辑器打开）。文件打开后，选择所有的内容，将其复制到剪贴板。

回到亚马逊的虚拟主机，切换到主目录，用nano文本编辑器打开一个新文件。

```
cd ~/
nano chapter11script.py
```

nano为命令行文本编辑器，上述命令将在命令行打开新文件。

文件打开后，把剪贴板的内容粘贴到里面。在有些系统中，你可能需要借助ssh程序的文件操作命令来完成文件上传，而不是按快捷键Ctrl+V粘贴。

在nano中，按下Ctrl+O保存文件，再按Ctrl+X退出nano。

你还需要字体文件。最简单的方法是从原地址再下载一遍，具体步骤如下。

```
wget http://openfontlibrary.org/assets/downloads/bretan/680bc56bbeeca9535
3ede363a3744fdf/bretan.zip
sudo apt-get install unzip
unzip -p bretan.zip Coval.otf > Coval.otf
```

上述代码用`unzip`命令解压Coval.otf文件（压缩包里有大量的zip文件，我们用不到）。

在虚拟机的命令行里输入以下命令运行程序。

```
python3 chapter11script.py
```

程序将会像在IPython Notebook笔记本里那样运行，把结果输出到命令行。

结果应该跟之前一致，但是神经网络的训练和测试比之前更快。程序其他方面可能没那么快——生成验证码数据集部分没有使用GPU，因此速度不会提升。

> 你可能希望关闭亚马逊虚拟主机来省点钱，本章最后运行大实验的相关程序时会再次用到虚拟机，我们接下来先在本地计算机上进行开发。

11.6 应用

回到本地计算机，打开本章创建的笔记本文件——我们用来加载CIFAR数据集的。接下来的大实验将使用CIFAR数据集，创建深度卷积神经网络，然后在虚拟机上使用GPU来运行。

11.6.1 获取数据

首先，用CIFAR图像创建数据集。跟之前不同的是，保留像素结构——像素的行列号。先把所有批次的图像文件名存储到列表中。

```
import numpy as np
batches = []
for i in range(1, 6):
    batch_filename = os.path.join(data_folder, "data_batch_{}".
format(i))
    batches.append(unpickle(batch1_filename))
    break
```

上面最后一行的`break`语句是用来测试代码——这会极大地减少训练数据，便于你迅速了解代码是否能正常运行。测试过代码能正常工作后，我会提示你删掉这一行。

接着，把每批次的图像文件依次添加到数据集里。我们用到了NumPy的`vstack`方法，你可以把它看作是往数组末尾追加一行数据。

```
X = np.vstack([batch['data'] for batch in batches])
```

然后，把像素值归一化，并将其强制转换为32位浮点型数据（这是使用GPU进行计算的虚拟机唯一支持的数据类型）。

```
X = np.array(X) / X.max()
X = X.astype(np.float32)
```

类别数据处理方法相同，只不过我们使用hstack，它为数组末尾追加一列数据。然后使用OneHotEncoder把它转换为只含有一位有效编码的数组。

```
from sklearn.preprocessing import OneHotEncoder
y = np.hstack(batch['labels'] for batch in batches).flatten()
y = OneHotEncoder().fit_transform(y.reshape(y.shape[0],1)).todense()
y = y.astype(np.float32)
```

接下来，把数据集切分为训练集和测试集。

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_
size=0.2)
```

调整数组形状以保留原始图像的数据结构。图像原本是32像素见方，每个像素由三个值组成（分别表示红、绿、蓝的颜色值）。

```
X_train = X_train.reshape(-1, 3, 32, 32)
X_test = X_test.reshape(-1, 3, 32, 32)
```

我们现在准备好了训练集、测试集以及目标类别，可以创建分类器了。

11.6.2　创建神经网络

我们使用nolearn库创建神经网络，步骤跟我们上面重做第8章实验时所用到的一致。

首先，创建神经网络的各层。

```
from lasagne import layers
layers=[
        ('input', layers.InputLayer),
        ('conv1', layers.Conv2DLayer),
        ('pool1', layers.MaxPool2DLayer),
        ('conv2', layers.Conv2DLayer),
        ('pool2', layers.MaxPool2DLayer),
        ('conv3', layers.Conv2DLayer),
        ('pool3', layers.MaxPool2DLayer),
        ('hidden4', layers.DenseLayer),
        ('hidden5', layers.DenseLayer),
        ('output', layers.DenseLayer),
        ]
```

最后三层使用密集层，但是前面使用三组卷积层和池化层。此外，我们（必须）以输入层开

始。这样一共有10层。输入数据跟数据集同型，而不只跟输入层的神经元数量相等，输入层和输出层大小仍由数据集来定，跟前面讲过的相同。

开始创建神经网络（注意下面第二行代码没有结束，先不要加右半边括号）。

```
from nolearn.lasagne import NeuralNet
nnet = NeuralNet(layers=layers,
```

指定输入数据的形状，跟数据集形状一致（每个像素由三个值组成，图像32像素见方）。第一个值None表示`nolearn`每次使用默认数量的图像数据进行训练——它将立即用这些数据进行训练，降低算法运行时间。设置为None，避免硬编码，算法使用起来更灵活。

```
input_shape=(None, 3, 32, 32),
```

> 如果你想每次使用不同的数据量进行训练，就需要创建`BatchIterator`实例。感兴趣的读者可参考如下文件：https://github.com/dnouri/nolearn/blob/master/nolearn/lasagne.py，了解`NeuralNet`类中`batch_iterator_train`和`batch_iterator_test`参数是如何设置的。

接着，设置卷积层的大小。没有严格的规则可遵守，但是我发现一开始用下面这些值就不错。

```
conv1_num_filters=32,
conv1_filter_size=(3, 3),
conv2_num_filters=64,
conv2_filter_size=(2, 2),
conv3_num_filters=128,
conv3_filter_size=(2, 2),
```

`filter_size`参数规定卷积层用于查看图像窗口的大小。此外，还要设置池化层的大小。

```
pool1_ds=(2,2),
pool2_ds=(2,2),
pool3_ds=(2,2),
```

然后，设置两层隐含层（倒数第三层和倒数第二层）、输出层的大小，输出层大小跟数据集类别数量相等。

```
hidden4_num_units=500,
hidden5_num_units=500,
output_num_units=10,
```

最后一层需要设置非线性函数，还是用`softmax`。

```
output_nonlinearity=softmax,
```

还要设置学习速率和冲量。据经验来看，随着数据量的增加，学习速率应该下降。

```
update_learning_rate=0.01,
update_momentum=0.9,
```

跟之前一样，把regression参数设置为True，训练步数设置为很小的值3，因为我们创建的这个神经网络运行时间相对较长。神经网络确定能正常运行之后，可以尝试增加训练步数以得到更好的模型，但是可能要花一两天时间（甚至更长！）来训练它。

```
                regression=True,
                max_epochs=3,
```

最后一个参数verbose设置为1，这样每步训练都会输出结果，以便于我们了解模型的训练进度以及它是否在运行。此外，还能输出每一步训练所花的时间，这个值变化不大，因此用每步训练所花时间乘以剩余训练步数就能算出总共还需要多长时间才能完成训练。

```
                verbose=1)
```

11.6.3 组装起来

现在就可以在训练集上训练我们刚创建的神经网络。

```
nnet.fit(X_train, y_train)
```

这可真是要费点时间了，即使我们只使用了部分训练数据，并且还限制了训练步数。一旦代码运行结束，测试方法跟之前一样。

```
from sklearn.metrics import f1_score
y_pred = nnet.predict(X_test)
print(f1_score(y_test.argmax(axis=1), y_pred.argmax(axis=1)))
```

结果很糟糕——本应如此！我们没有经过充分训练——只进行了三轮迭代，且只使用了1/5的数据。

首先，回过头去删除创建数据集代码中的break语句（在遍历每批数据的for循环中）。这样就能使用所有数据而不只是部分数据进行训练。

接着，在神经网络的定义中，把训练步数改为100。

现在，把代码粘贴到虚拟机上。跟之前一样，点击File | Download as，选择Python，把代码保存为.py文件。启动、连接到虚拟机，像之前把代码那样粘贴过去（我把这里的.py文件命名为chapter11cifar.py——如果你使用其他名字，请记得在下面代码中做相应改动）。

接下来，我们需要把数据集捣腾到虚拟机上。最简单的方法是在虚拟机的命令行输入以下命令。

```
wget http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
```

下载数据集，新建Data文件夹，把数据文件解压到Data文件夹中。

```
mkdir Data
tar -zxf cifar-10-python.tar.gz -C Data
```

最后，使用下面命令运行实验代码。

```
python3 chapter11cifar.py
```

你首先注意到的将是速度大幅提升。在我的本地计算机上，每步训练需要100秒以上。在启用GPU的虚拟机上，每步训练只需16秒！在我本地计算机上进行100步训练，需要近3个小时，而虚拟机只需要26分钟。

速度上的大幅提升使得我们可以快速测试不同模型的效果。对于机器学习算法调试来说，一个算法的计算复杂性问题不大，只要几秒钟、几分钟，多则几个小时就可以运行完。因此，只有一个模型，训练时间长点短点不会有太大问题——特别是大多数机器学习算法都能很快给出预测结果，这也正是为什么大多数情况都使用一个模型。

然而，要调试多个参数时，你就需要训练成千上万个模型，各模型之间参数差异很小——速度提升问题就变得很重要。

26分钟后，完成100步训练，得到最终的输出结果。

```
0.8497
```

结果还不错！我们可以增加训练步数，进一步改善效果或者也可以尝试修改参数值，增加隐含层节点、卷积层的数量，或者多使用一层密集层。也可以尝试用Lasagne其他类型的层，虽然一般来讲，卷积层更适合处理视觉问题。

11.7 小结

本章讲解了深度神经网络，为了处理计算机视觉问题，着重讲解卷积网络。我们使用Lasagne和nolearn库来构建神经网络，很多数据处理工作交由Theano来做。用nolearn提供的工具构建神经网络相对比较容易。

卷积神经网络专门用来处理计算机视觉问题，所以最后得到的分类结果很精确也不足为奇。最终结果表明，随着算法和计算能力的提升，计算机视觉的应用前景也更为广阔。

我们在虚拟机上使用GPU计算，大幅提升训练速度，在虚拟机上运行的速度几乎是我本地计算机的10倍。如果某些算法需要额外的计算能力，可以考虑使用性价比很高的云主机（一般每小时不到1美元）——只要记得用完后即时关机就好！

本章所研究的卷积网络算法相当复杂。卷积网络训练时间长，有很多参数需要训练。数据集看似不小，其实还称不上大数据，甚至不用稀疏矩阵，就能把这些数据全部加载到内存。下章所讲的算法更加简单，但是数据集却特别大，无法装进内存，这是大数据最显著的一个特点，采矿业和社交网络等行业所产生的数据都具有这个特点。

第 12 章

大数据处理

12

当今社会数据量呈指数级增长，这些数据来自于用户行为监测系统、分布式系统、网络分析、传感器等。移动互联网的迅速发展又带来移动数据的增长，此外，下一个大潮流——物联网（Internet of Things，IoT）又会进一步提升数据增长速度。

挖掘大数据需要有新思路。运行时间很长的复杂算法要么改进要么抛弃，而能够处理更多数据，复杂程度相对简单的算法正变得更加流行起来。例如，支持向量机是很好的分类器，但是它的一些变种很难在大型数据集上使用。相反，逻辑回归等相对简单的算法处理大数据集更容易。

本章主要内容如下:

- 大数据的挑战及其应用
- MapReduce范式
- Hadoop MapReduce
- 在亚马逊平台上运行MapReduce程序的Python包mrjob

12.1　大数据

大数据具有哪些特点呢？它的大多数支持者认为有以下4个特点，简称4V。

(1) 海量（Volume）：我们产生和存储的数据量加速增长，未来势头看起来会更猛。现在常用的硬盘容量单位为GB，再过几年就会变为EB[①]。同时网络吞吐量也会增长。信噪比不容乐观，重要信息很可能被海量重要程度低的信息湮没。

(2) 高速（Velocity）：数据量增长的同时，数据处理速度在加快。就拿现如今的汽车举例子，每辆车装有成百上千个传感器，这些传感器不停地向汽车内置的计算机传送数据，要完成操纵汽车的任务，数据的分析处理速度需要达到次秒级水准。这不仅要从大量数据中寻找答案，还得速度快。

① EB（exabyte），艾字节，1EB=1024PB，1PB=1024TB，1TB=1024GB。1EB约等于10亿GB。——译者注

(3) 多样（Variety）：数据集有多种形式，各列清晰定义的规整数据集只占其中很少一部分。想想社交网站的一条消息吧，它可能包含文本、照片、提及其他用户[①]、喜欢数、评论数、视频、地理位置信息等。简单地忽略掉这些难以整合进模型的数据势必会导致信息丢失。

(4) 准确（Veracity）：随着数据量的增加，很难确定采集到的数据是否正确——是否过时、充满噪音，有没有包含异常数据，或者总体来说是否有用等。倘若人们无法对数据的准确性进行有效验证，要信赖数据很难。人们从外部得到的数据集被不停地整合到内部数据集中，使得数据的准确性这个问题更加突出。

从以上四点（也有人认为是五点[②]）就能看出大数据不仅仅是大量的数据，处理大数据对工程能力要求很高——更不用说对它们进行分析了。很多"蛇油商人"[③]只关心怎样夸大大数据的应用，却避而不谈大数据工程的挑战以及潜在分析工作的难度。

我们前面用过的这些算法都是把数据集加载到内存后再进行处理。这对于提升计算速度很有好处，因为处理已经在内存中的数据比起先从硬盘加载到内存后再处理要快得多。此外，我们可以对内存中的数据进行多次迭代，改善模型。

对于大数据，我们就无法将其加载到内存中。从很多方面来讲，可以用此来判断一个问题能不能称得上大数据问题——如果数据可以加载到你计算机内存中，那就算不上大数据。

12.2 大数据应用场景和目标

企业和个人都离不开大数据。

所有以大数据为基础的系统中，人们与之打交道最多的当属谷歌这样的搜索引擎。对于搜索引擎来说，要在零点几秒的时间内对几十亿甚至更多的网站完成一次搜索，基础的文本查询肯定满足不了需求，单是存储这么多网站的文本内容就是个大难题。要完成搜索引擎中的查询任务，就需要专门创建新数据结构和使用新数据挖掘方法。

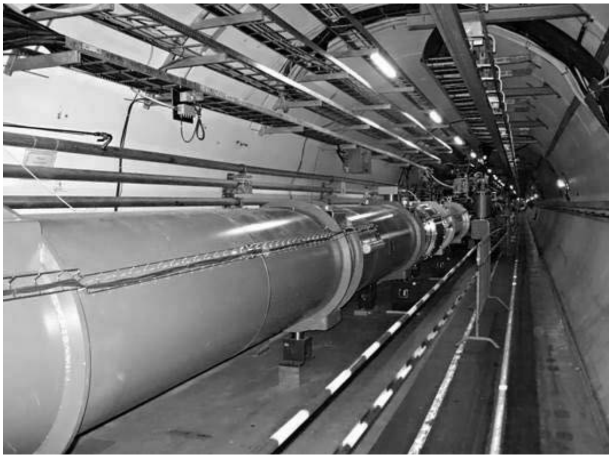
很多其他科学实验也都用到了大数据技术，例如大型强子对撞机（Large Hadron Collider）。该机器长达27公里，装有1.5亿个传感器，用于监测每秒几亿个强子的对撞情况，对撞机局部照片请见下图。对撞实验产生的数据量十分惊人，每天过滤后还有25PB（假如不过滤，每年将产生1.5亿PB数据）。这些数据隐藏着宇宙的奥秘，对其进行分析能加深我们对宇宙的认识，但是工

① 指"@"加用户名这种形式。——译者注

② 有人认为大数据的另一个特点是价值（Value），经过挖掘得到的结论有实际应用价值。——译者注

③ 蛇油商人（snake oil salesman）跟"江湖郎中"有相同的贬义内涵。19世纪，国人赴美修建北太平洋铁路，带去从南方水蛇体内提炼的蛇油，以祛风湿，缓筋骨之痛，颇受当地人欢迎。后美国商人Clark Stanley等改从北美响尾蛇提炼蛇油，批量生产，但效果较差，甚至出现产品中压根不含蛇油的造假现象。该词遂被用来指打着不容置疑的旗号，推销假冒伪劣谋求私利的骗子。——译者注

程量很大，分析起来也相当困难。



政府部门对大数据的重视程度也在逐渐增加，以充分了解人口、商业活动和国家其他各方面的情况。跟踪几千万乃至几亿人口及成百上千亿的交易量（商贸或医疗开支），很多政府机构都需要依靠大数据分析技术才能完成。

全球很多政府尤其关注交通管理，他们使用数以千万计的传感器判断哪些道路堵塞最为严重，预测新修道路对交通状况的影响。

大型零售商使用大数据改善客户体验，降低支出。这包括预测客户需求，保证供货量适度，向客户推销他们可能喜欢的产品，跟踪交易数据，寻找购物潮流和消费模式，发现潜在欺诈行为。

还有很多大公司用大数据提高公司经营管理的自动化程度，改善产品和服务质量。他们通过大数据分析预测行业风向，跟踪外部竞争者。他们还把大数据用到员工管理上——跟踪员工，发现其离职倾向以及时干预。

信息安全部门通过监测网络流量，结合大数据技术，寻找大型网络的恶意软件感染，这包括寻找奇怪的流量模式，恶意软件传播迹象和其他反常现象。高级持续性威胁（Advanced Persistent Threats，APTs）攻击也很令人抓狂，攻击者把恶意代码藏在大型网络中，长期从网络中窃取信息或从事其他破坏活动。寻找APTs往往需要检查很多台计算机进行取证，费时费力，仅靠人工很难完成，这时就可以使用大数据技术实现自动化检查、取证。

大数据方兴未艾，正在越来越多的行业和应用中大显身手。

12.3　MapReduce

大数据挖掘和计算有几个主要概念，其中最为流行的就是MapReduce模型，它可用于任意大

数据集的一般性计算任务。

谷歌出于并行计算的需要，最先提出了MapReduce模型。它还引入了容错和可伸缩特性。MapReduce最初的研究发表于2004年，打那时起，成千上万的项目、实现和应用都用到了这一概念。

虽然MapReduce跟之前很多概念有相似之处，但这不影响它成为大数据分析领域的核心概念。

12.3.1 直观理解

MapReduce主要分为两步：映射（Map）和规约（Reduce）。函数式程序设计中有把函数映射到列表和规约结果的概念，MapReduce以这两个概念为基础。为了解释映射和规约，我们举个例子，编写代码遍历一串列表，计算所有列表中数字之和。

MapReduce范式还包括排序（shuffle）和合并（combine）两步，后面会讲。

首先，在映射这一步，接收一个函数，用这个函数处理列表的各元素，返回跟之前列表长度相等的列表，新列表的元素为函数的返回结果。

打开一个新的IPython Notebook笔记本文件，创建一列表，列表中各元素为包含几个数字的列表。

```
a = [[1,2,1], [3,2], [4,9,1,0,2]]
```

接着，建立sum函数和a之间的映射关系。这一步会用sum函数处理列表a的每一个元素。

```
sums = map(sum, a)
```

上述sums为生成器（在我们调用它之前，不会进行计算），上面这行代码大体上等价于：

```
sums = []
for sublist in a:
    results = sum(sublist)
    sums.append(results)
```

规约步骤要稍微复杂些，需要对返回结果的每一个元素应用一个函数，从初始值开始，对初始值和第一个值应用指定函数，得到返回结果，然后再对所得到的结果和下一个值应用指定函数，以此类推。

我们来创建一个函数，接收两个数字作为参数，返回两个数字的和。

```
def add(a, b):
  return a + b
```

然后进行规约。规约函数形式为reduce(function, sequence, initial)，参数分别表示用来进行规约的函数的名字、列表和初始值，函数即是对序列的每一步所应用的函数。在第一步，第一个值为初始值，而不是列表的第一个元素。

```
from functools import reduce
print(reduce(add, sums, 0))
```

结果为25，它是sums列表中每个元素的和，也就是原来大列表a中每个二级列表各元素的和。

上面代码等价于如下代码。

```
initial = 0
current_result = initial
for element in sums:
    current_result = add(current_result, element)
```

我们这个小例子，代码很简短，但真正的好处是可以使用分布式计算。假如，二级列表的数量为100万，每个二级列表包含100万个元素，对于这么大的计算任务，我们可以交由多台计算机完成。

为了实现分布式计算，我们可以在映射这一步把各个二级列表及函数说明分发到不同的计算机上。计算完成后，各计算机把结果返回主计算机（master）。

然后master把结果发送给另一台计算机做规约。我们的例子有100万个二级列表，因此可以将100万个任务交给不同的计算机处理（计算机完成映射操作后，可再次用于规约）。返回结果为包含100万数字的列表，然后就可以进行求和运算。

这样做的好处是，尽管原始数据集有1万亿数字，但实际计算过程，哪台计算机都不需要存储超过100万个数字。

12.3.2　单词统计示例

MapReduce的实现比起单纯使用映射和规约两步要复杂些。这两步都用键来调用，便于数据的区分和值的跟踪。

映射函数接收一键值对，返回键值对列表。接收和输出的键不一定要彼此相关。例如，统计单词数量的MapReduce程序，输入的键可能是文档编号，而输出的键却可以是单词。输入值可能是文档的文本内容，而输出值为每个单词的词频。

```
from collections import defaultdict
def map_word_count(document_id, document):
```

首先，计算每个单词词频。在这个简化过的例子中，我们先根据空格把文档转换成单词列表，当然有更好的解决方法。

```
    counts = defaultdict(int)
    for word in document.split():
        counts[word] += 1
```

遍历每个单词，统计次数，注意用到了yield语句。用MapReduce的术语来说，单词为键，单词出现次数为值。

```
    for word in counts:
        yield (word, counts[word])
```

用单词做键，我们就可以进行shuffle操作，把每个键所有值聚集到一起。

```
def shuffle_words(results):
```

首先，把每个单词的统计结果放到一个列表中。

```
    records = defaultdict(list)
```

遍历映射函数返回的所有结果。

```
    for results in results_generators:
        for word, count in results:
            records[word].append(count)
```

接着，遍历每个单词，创建生成器，它能生成（yield）单词和该单词在各文档出现次数的列表这两项。

```
    for word in records:
        yield (word, records[word])
```

最后一步是规约，接收一键值对（值为列表），输出另外一键值对。我们这里，键为单词，输入的列表为shuffle后得到的作为键的单词在不同文档出现次数的列表，输出键值对中的值这一项为单词（键）在所有文档的出现次数之和。

```
def reduce_counts(word, list_of_counts):
    return (word, sum(list_of_counts))
```

我们使用scikit-learn提供的来自20个新闻组的语料看下上述代码的实际效果。

```
from sklearn.datasets import fetch_20newsgroups
dataset = fetch_20newsgroups(subset='train')
documents = dataset.data
```

然后执行映射操作，这里用enumerate函数自动为文档生成编号。编号这里用处不大，但是在其他应用中很重要。

```
map_results = map(map_word_count, enumerate(documents))
```

上述结果只是生成器，而不是实际的统计结果。也就说，它是能输出键值对（单词、出现次数）的生成器。

接着，对生成器进行shuffle操作，根据单词出现次数进行排序。

```
shuffle_results = shuffle_words(map_results)
```

上述例子本质上来说是一个MapReduce任务；然而由于只使用单线程，我们无法从MapReduce数据格式中获得任何好处。下节，我们将使用开源的MapReduce架构Hadoop，实现分布式计算，提升性能。

12.3.3　Hadoop MapReduce

Hadoop是Apache基金会所提供的一组包括MapReduce在内的开源工具。人们实际使用的MapReduce架构也多是它。Hadoop项目由Apache基金会管理（他们也负责维护著名的Web服务器软件Apache）。

Hadoop生态系统很复杂，有大量的工具。我们将要用到的主要组件为Hadoop MapReduce。Hadoop其他处理大数据的工具有如下几种。

- Hadoop分布式文件系统（Hadoop Distributed File System，HDFS）：该文件系统可以将文件保存到多台计算机上，以防范硬件故障，提高带宽。
- YARN：用于调度应用和管理计算机集群。
- Pig：用于MapReduce的高级语言。Hadoop MapReduce用Java语言实现，Pig对Java实现做进一步封装，支持用其他语言来编写程序——包括Python。
- Hive：用于管理数据仓库和进行查询。
- HBase：对谷歌分布式数据库BigTable的一种实现。

这些工具都是用来解决包括分析过程在内的大数据实验所能遇到的各种问题。

还有其他一些没有使用Hadoop工具集实现的MapReduce框架，也有一些具有相似目标的项目。此外，很多云平台都提供以MapReduce为基础的系统。

12.4　应用

我们来建立一个根据博主用词习惯判断博主性别的应用。我们用MapReduce方法训练朴素贝叶斯分类器。最后用模型做预测时，不需要MapReduce，虽然我们可以用映射这个步骤——也就是对列表中的每一篇文档运行预测模型。这是用MapReduce进行数据挖掘常用的映射操作，规约步骤只用来调整预测结果列表，以便把结果和原文档对应起来。

我们将使用亚马逊云平台运行应用，以利用它们的计算资源。

12.4.1 获取数据

我们使用2004年8月份从http://blogger.com网站采集的60多万篇博客文章作为数据集，这些文章共有1.4亿多个单词，每篇文章都标注了博主的年龄、性别、行业（工作），有趣的是还有星座。研究人员曾进行过部分验证，确保同一篇博客从头到尾均由一名博主所写（虽然也不是很确定）。每篇博客还给出了发表时间，这个数据集的内容真是挺丰富的。

访问http://u.cs.biu.ac.il/~koppel/BlogCorpus.htm网站，点击Download Corpus。下载完成后，把数据集文件解压到你的本地计算机数据文件夹Data里。

数据集中，一个博主的所有博客放到同一个文件里，文件名包含博主的相关信息。例如，其中一个文件名如下：

```
1005545.male.25.Engineering.Sagittarius.xml
```

文件名用点号分隔，主要包括以下信息。

- 博客编号：标识博客唯一性的数值。
- 性别：该部分不是male（男性）就是female（女性）。数据集只包含这两种情况。
- 年龄：给出确切的年龄，但有时会特意使用年龄段。年龄段（包含起止年龄）有13~17、23~27和33~48。这样做是便于把难以确定博主年龄的博客粗略归到某一年龄段中，因为很难把18岁写的博客和19岁所写的区分开来，此外，当你使用这些数据时，博主实际年龄可能比早先填写的又长了几岁，因此需要做相应调整。[①]
- 行业：包括科学、工程、艺术、房地产在内的40种行业的其中一种，若博主没有填写该项，使用indUnk[②]来表示。
- 星座：12星座之一。

所有的这些数据都是用户自己提供的，这就表明其中有错误或不一致现象，但是绝大多数是可靠的——如果用户为了保护个人隐私起见，不想透露某方面的信息，他们可以选择不填，这一点网站是允许的。

每个文件的格式类似于XML，包含`<Blog>`标签和一系列`<post>`标签。每个`<post>`标签前都有一个`<date>`标签。虽然我们可以把它当作XML来处理，但是按行处理更容易，因为它不是真正规范的XML文件，并且还有些错误（大部分是编码错误）。我们可以使用循环结构遍历文件中的每一行，以读取博客内容。

指定一个文件名，我们来实际看下。

① 作者给我举了个例子，“去年用户发表博客时为18岁，今年就19岁了，我需要在模型中做相应调整。”——译者注
② industry unknown的简写。——译者注

```
import os
filename = os.path.join(os.path.expanduser("~"), "Data", "blogs",
"1005545.male.25.Engineering.Sagittarius.xml")
```

首先，创建用于存储每篇博客的列表。

```
all_posts = []
```

然后，打开要读取的文件。

```
with open(filename) as inf:
```

设置标识是否在博客中的标记。找到博客开始标签<post>后，将标记的值设置为True，表示找到了博客的开始位置。找到关闭标签</post>后，将标记的值设置为False。

```
    post_start = False
```

创建用于存储博客当前行内容的列表。

```
    post = []
```

遍历文件的每一行，删除该行前后的空格。

```
    for line in inf:
        line = line.strip()
```

照前面所说，找到博客的开始、关闭标签后，更改标记post_start的值。

```
        if line == "<post>":
            post_start = True
        elif line == "</post>":
            post_start = False
```

找到关闭标签</post>后，记录刚找到的这一篇博客的所有内容。创建新的post列表。下面两行代码的缩进字符数与前一行代码相同。

```
            all_posts.append("\n".join(post))
            post = []
```

最后，如果该行不是博客结尾，也就是还在博客里面，把这一行加到当前博客post列表的最后。

```
        elif post_start:
            post.append(line)
```

如果没有在博客中，忽略这一行。

可以像下面这样获取每篇博客的文本内容。

```
print(all_posts[0])
```

我们还可以计算出一位博主共发表了多少博客。

```
print(len(all_posts))
```

12.4.2 朴素贝叶斯预测

我们来实现朴素贝叶斯算法（从技术上讲，这是一个简化的版本，复杂实现通常有很多特征）处理博客博主性别分类问题。

1. mrjob包

把用mrjob创建的MapReduce任务推送到亚马逊云平台上很容易。mrjob听上去有点像奇先生①系列儿童故事书的画蛇添足之作，实则表示映射规约任务（Map Reduce Job）。这个包很有用，但是写作本书时，对Python 3的支持还不成熟，对我们后面要讲到的亚马逊EMR服务支持也不够好。

> 你可以使用下面命令安装mrjob的Python 2版本：
>
> **sudo pip2 install mrjob**
>
> 注意使用pip2而不是pip3。

本质上讲，mrjob提供了大部分MapReduce任务所需的标准功能。最令人惊异的特性是，你可以编写同一套代码，既能在没有安装Hadoop的本地计算机上进行测试，测试完后，还可以直接把代码提交到亚马逊的EMR或其他Hadoop服务器上。

这样，测试起代码来很容易，虽然它也不可能神奇地把大问题变成小问题——注意任何本地测试，只是使用一小部分而不是全部数据。

2. 抽取博客内容

首先创建一个MapReduce程序，从每位博主的博客文件中抽取所有博客，分别存储。因为最终要预测博主的性别，我们还需要抽取性别信息，跟博客一起存储。

我们这回没法用笔记本文件，因此使用Python IDE进行开发。如果你没有装Python IDE（比如PyCharm），你可以使用文本编辑器。建议你使用具有代码高亮功能的IDE。

> 如果你找不到好用的IDE，你可以在笔记本文件中编写代码，然后点击（或选择）File | Download As | Python把代码保存成.py文件，然后再运行。第11章提到过这种方法。

① 英国儿童文学作家罗杰·哈格里维斯（Roger Hargreaves）为儿童创作的《奇先生妙小姐》系列图书中的人物形象，比如有荒唐先生、傲慢先生、邋遢先生等。——译者注

因为需要获取环境变量，我们会用到os，单词拆分时会用到正则表达式，一并导入re。

```
import os
import re
```

接着导入MRJob类，我们的MapReduce任务继承自它。

```
from mrjob.job import MRJob
```

然后创建MRJob的子类。

```
class ExtractPosts(MRJob):
```

我们使用跟之前类似的循环从文件中抽取博客内容。我们即将定义的映射函数将处理每一行，跟踪不同博客的任务要在映射函数外完成。因此，在函数外而不是在函数内部声明post_start和post两个变量。

```
    post_start = False
    post = []
```

然后创建映射函数——从文件中取一行作为输入，最后生成一篇博客的所有内容。每一行都来自同一任务所在处理的文件。这样，我们就可以使用上面创建的变量保存当前博客的内容。

```
    def mapper(self, key, line):
```

开始采集博客内容之前，我们还要获取到博主的性别。虽然通常我们不会把文件名作为MapReduce任务的一部分，但是这里确实要用到。当前的文件名称以环境变量的形式存储，用下面代码就能获取到。

```
        filename = os.environ["map_input_file"]
```

用点号切分文件名，获取性别（第二个字符串）。

```
        gender = filename.split(".")[1]
```

删除一行开头和结尾处的空格（博客中空格很多），然后查找文件开头和结尾。

```
        line = line.strip()
        if line == "<post>":
            self.post_start = True
        elif line == "</post>":
            self.post_start = False
```

之前我们把博客保存到列表里，而这次我们用yield，便于mrjob跟踪输出：博主性别和博客内容，这样博主性别和博客就能一一对应起来。函数余下部分跟前面用到的循环中的代码一致。

```
            yield gender, repr("\n".join(self.post))
            self.post = []
        elif self.post_start:
            self.post.append(line)
```

最后，在函数和类的外面，编写下面语句，以便从命令行运行代码时执行MapReduce任务。

```
if __name__ == '__main__':
    ExtractPosts.run()
```

现在，可以在命令行输入下面的shell命令运行MapReduce任务。注意使用Python 2而不是Python 3。

```
python extract_posts.py <your_data_folder>/blogs/51* --output-
  dir=<your_data_folder>/blogposts -no-output
```

第一个参数`<your_data_folder>/blogs/51*`（注意把`<your_data_folder>`替换为你的数据文件夹的全路径）指定所使用的数据集（所有以51开始的文件，只有11个）。接着指定输出文件夹，即用来保存博客内容的文件夹。最后指定不要在命令行输出内容。否则，程序运行期间，输出的数据将显示在命令行——对我们来说没多大用处，还会降低程序运行速度。

运行上述代码，每篇博客内容很快就会被抽取出来并保存到指定的输出文件夹。在本地计算机上，只使用一个线程，所以速度提升有限，但是我们知道代码确实可以运行了。

我们看下输出文件夹，里面有有一系列文件，文件的每一行为博主的性别及一篇博客的内容。

3. 训练朴素贝叶斯分类器

既然已经抽取了博客内容，接下来就可以用它们训练朴素贝叶斯模型。根据直觉，我们可以这样做，分别统计女性博主和男性博主使用每个单词的概率。对博客进行分类时，我们分别计算博客[①]博主为女性的概率和博主为男性的概率，选取概率较大的，作为最终类别。

我们来编写代码，输出所有文件中每个单词及女博主和男博主使用该词的频率。输出文件如下所示：

```
"'ailleurs"  {"female": 0.003205128205128205}
"'air"  {"female": 0.003205128205128205}
"'an"  {"male": 0.0030581039755351682, "female": 0.004273504273504274}
"'angoisse"  {"female": 0.003205128205128205}
"'apprendra"  {"male": 0.0013047113868622459, "female":
0.0014172668603481887}
"'attendent"  {"female": 0.00641025641025641}
"'autistic"  {"male": 0.002150537634408602}
"'auto"  {"female": 0.003205128205128205}
"'avais"  {"female": 0.00641025641025641}
"'avait"  {"female": 0.004273504273504274}
"'behind"  {"male": 0.0024390243902439024}
"'bout"  {"female": 0.002034152292059272}
```

① 一篇博客，其博主为女性的概率有多大？计算方法为，从语料中统计到该博客中每个单词出现在女博主所写博客中的概率，然后求出这些概率的积。其中会涉及到数值下溢、博客中某一单词在语料中没有出现等问题，这些问题需要特殊处理。下文也会讲到。——译者注

输出结果每一行的第一个值为单词，第二值为一个字典，字典的键为性别，字典的值为给定性别使用该词的频率。

用Python IDE或文本编辑器新建一个文件，再次要用到os和re，除此之外还要用到NumPy和MRJob。因为要对字典进行排序，所以还用到itemgetter。

```
import os
import re
import numpy as np
from mrjob.job import MRJob
from operator import itemgetter
```

我们还需要用MRStep管理MapReduce中的每一步操作。前面的MapReduce任务只有由映射函数和规约函数组成的一个步骤。我们当前这个任务分三步：映射、规约、再次映射和规约。是不是感觉跟前几章我们用到的流水线一样，前一步的输出将作为下一步的输入。

```
from mrjob.step import MRStep
```

接着创建用于匹配单词的正则表达式，并对其进行编译，我们用它来查找单词的边界。这种类型的正则表达式比起前面用过的split方法更加强大，如果你需要更加精确的单词切分工具，建议你使用第6章用到的NLTK库。

```
word_search_re = re.compile(r"[\w']+")
```

创建一个新类，用于训练朴素贝叶斯分类器。

```
class NaiveBayesTrainer(MRJob):
```

定义MapReduce任务的各个步骤，一共分为两步。第一步抽取单词出现的概率。第二步，比较一个单词在男女博主所写博客出现的概率，选择概率较大的性别作为分类结果，写入输出文件。在上述每一步（MRStep）中，定义映射和规约函数，它们是NaiveBayesTrainer类里面的方法（后续会编写这两个函数）。

```
    def steps(self):
        return [
            MRStep(mapper=self.extract_words_mapping,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.compare_words_reducer),
            ]
```

第一个函数是第一步中的映射函数。这个函数的目标是接收一条博客数据，获取里面的所有单词，因为我们想要得到单词的频率，所以每个单词返回1 / len(all_words)，便于后面加总求词频。这样计算并不精确——我们需要根据文档数量做归一化处理。但由于数据集中两个类别的文档数相同，因此不做归一化处理对最终结果影响也很小。

输出博主的性别，后面会用到。

```
    def extract_words_mapping(self, key, value):
        tokens = value.split()
        gender = eval(tokens[0])
        blog_post = eval(" ".join(tokens[1:]))
        all_words = word_search_re.findall(blog_post)
        all_words = [word.lower() for word in all_words]
            all_words = word_search_re.findall(blog_post)
            all_words = [word.lower() for word in all_words]
            for word in all_words:
                yield (gender, word), 1. / len(all_words)
```

> 上面代码使用eval函数简化对每条博客数据的处理，直接把字符串转换为Python列表，但不建议这么做。相反，应该使用JSON等格式存储数据，然后用相应包进行解析。如果在访问数据集的代码中使用eval语句，攻击者可借此插入恶意代码，并在你的服务器上运行。

在第一步的规约函数中，汇总每个性别使用每个单词的频率。我们还把键改为单词，而不是单词和性别的组合，因为在最后训练得到的模型中，我们要根据单词进行查询（虽然还要输出性别以供后面使用）。

```
    def reducer_count_words(self, key, frequencies):
        s = sum(frequencies)
        gender, word = key
        yield word, (gender, s)
```

最后一步不需要映射函数，因此我们就没有添加它。数据将作为一致性映射（identity mapper）类型直接传入到规约函数中，而规约函数将会把每个单词在所有文章中出现频率按照性别汇集到一起，输出单词及频率字典。

这正是朴素贝叶斯分类器所需要的信息。

```
    def compare_words_reducer(self, word, values):
        per_gender = {}
        for value in values:
            gender, s = value
            per_gender[gender] = s
        yield word, per_gender
```

最后，添加以下代码，以便直接运行代码时，训练朴素贝叶斯模型。

```
if __name__ == '__main__':
    NaiveBayesTrainer.run()
```

运行上述代码，它的输入即是前面博客抽取代码的输出（实际上可以把它们作为前后相连的步骤，放到同一个MapReduce任务中）。

```
python nb_train.py <your_data_folder>/blogposts/
  --output-dir=<your_data_folder>/models/
--no-output
```

上述代码运行结束后，该MapReduce任务的输出结果保存到位于输出文件夹中的一个文件里，输出结果为运行朴素贝叶斯分类器所需的概率信息。

4. 组装起来

我们现在就可以使用这些概率来运行贝叶斯分类器。我们将在笔记本文件里编写代码，可以再次使用Python 3喽！

首先，查看上一个MapReduce任务所生成的模型文件夹。如果输出文件多于一个，在命令行切换到模型文件夹下，使用下面命令把它们追加到model.txt后面。

```
cat * > model.txt
```

运行上述命令后，记得把下面用到的模型所在的文件名改为model.txt。

回到笔记本文件，导入接下来会用到的几个包。

```
import os
import re
import numpy as np
from collections import defaultdict
from operator import itemgetter
```

重新定义用于查找单词的正则表达式——在实际应用中，不要忘记这点，注意训练和测试时应该使用相同的正则表达式来抽取单词。

```
word_search_re = re.compile(r"[\w']+")
```

接着，声明从指定文件名中加载模型的函数。

```
def load_model(model_filename):
```

模型的参数是一个元素为字典的字典，各元素的键为单词，值（内部字典）为由每个性别及其概率组成的键值对。我们使用`defaultdict`，如果键不存在的话，将返回0。

```
    model = defaultdict(lambda: defaultdict(float))
```

打开模型所在文件，解析每一行。

```
    with open(model_filename) as inf:
        for line in inf:
```

用空格作为分隔符，将模型的每一行切分成两部分。第一部分为单词自身，第二部分为概率字典。对于每一部分，使用`eval`函数获得实际的值，它们之前是使用`repr`函数存储的。

```
                word, values = line.split(maxsplit=1)
                word = eval(word)
                values = eval(values)
```

在模型中，为单词和概率字典建立起映射关系。

```
            model[word] = values
    return model
```

接着，加载实际的模型。你可能需要修改模型的文件名——它存储在上一个MapReduce任务的输出文件夹中。

```
model_filename = os.path.join(os.path.expanduser("~"), "models",
"part-00000")
model = load_model(model_filename)
```

举例来说，我们可以像下面这样查看不同性别使用单词i（执行MapReduce任务时，所有单词中的字母全部转换为小写）的情况。

```
model["i"]["male"], model["i"]["female"]
```

接着，创建使用模型做预测的函数。这里我们不再使用`scikit-learn`接口，只是创建一个简单的函数。这个函数接收模型和一篇文档作为参数，返回对博主性别的预测结果，函数声明如下：

```
def nb_predict(model, document):
```

先来创建一个字典，键值分别为每个性别及概率①。

```
    probabilities = defaultdict(lambda : 1)
```

从文档中抽取每一个单词。

```
    words = word_search_re.findall(document)
```

遍历每一个单词，找出数据集中每个性别使用该单词的概率。

```
    for word in set(words):
        probabilities["male"] += np.log(model[word].get("male", 1e-
15))
        probabilities["female"] += np.log(model[word].get("female",
1e-15))
```

根据概率对性别进行排序，以概率最高的性别作为预测结果返回。

```
    most_likely_genders = sorted(probabilities.items(),
key=itemgetter(1), reverse=True)
    return most_likely_genders[0][0]
```

注意，我们使用`np.log`计算概率。朴素贝叶斯模型中，概率值往往很小。把这些很小的数连乘起来会导致数值下溢，由于计算机精度不够，最终结果将为0。在我们这里，就会出现博主

① 不同性别用词有自己的偏好。此处的概率指的是，待预测文档每一个单词由给定性别所使用的概率的乘积。
——译者注

为男性或女性的概率都为0的情况，预测结果自然不正确。

为了解决数值下溢问题，我们使用对数概率。对于两个数a和b，`log(a,b)`等于`log(a) + log(b)`。数值很小的数取对数，得到一个负数，但相对还比较大。例如`log(0.00001)`约等于–11.5。这表明与其冒着数值下溢的风险，把多个概率连乘，还不如使用对数概率求和，然后比较两类的和（和越大，可能性越高）。

使用对数概率有个问题就是无法处理0值（虽然，把多个为0的概率连乘也不能解决这个问题）。这是因为`log(0)`没有定义。有些朴素贝叶斯算法，为每个特征的计数都加1以避免出现这个问题，当然还有别的解决方法。加一平滑是一种很简单的数据平滑方法。我们这里如果某个单词给定性别没有人用过，就给它赋一个很小的概率值。

再回到我们的预测函数，我们从数据集中复制一篇博客来做下测试。

```
new_post = """ Every day should be a half day.  Took the afternoon
off to hit the dentist, and while I was out I managed to get my oil
changed, too.  Remember that business with my car dealership this
winter?  Well, consider this the epilogue.  The friendly fellas at the
Valvoline Instant Oil Change on Snelling were nice enough to notice
that my dipstick was broken, and the metal piece was too far down in
its little dipstick tube to pull out.  Looks like I'm going to need a
magnet.   Damn you, Kline Nissan, daaaaaaammmnnn yooouuuu....   Today
I let my boss know that I've submitted my Corps application.  The news
has been greeted by everyone in the company with a level of enthusiasm
that really floors me.     The back deck has finally been cleared off
by the construction company working on the place.  This company, for
anyone who's interested, consists mainly of one guy who spends his
days cursing at his crew of Spanish-speaking laborers.  Construction
of my deck began around the time Nixon was getting out of office.
"""
```

用下面代码进行分类。

```
nb_predict(model, new_post)
```

分类结果为男性，分类正确。当然，千万不要只用一条数据进行测试。我们只使用文件名以51打头的文档来训练模型，所以不要指望有太高的正确率。

当务之急是用更多的文档来训练模型。我们使用所有文件名以6或7打头的文档做测试，在剩余文档上进行训练。

使用命令行，切换到博客文档所在的文件夹（`cd <your_data_folder>`），复制文档到新文件夹中。

创建训练集文件夹。

```
mkdir blogs_train
```

然后，创建测试集文件夹。

```
mkdir blogs_test
```

把所有文件名以6或7打头的文档从训练集文件夹移动到测试文件夹。

```
cp blogs/6* blogs_train/
cp blogs/7* blogs_train/
```

我们需要返回训练集中所有博客的抽取结果。然而，计算量很大，最好使用云平台而不是本地计算机处理这个任务。因此，我们接下来使用亚马逊云平台解析博客内容。

跟之前一样，在命令行运行下面代码。唯一的不同点是训练集所在的文件夹路径。运行代码前，删除用于存储抽取结果和模型文件夹中的所有文件。

```
python extract_posts.py ~/Data/blogs_train --output-dir=/home/bob/
Data/blogposts -no-output
python nb_train.py ~/Data/blogposts/ --output-dir=/home/bob/models/
--no-output
```

上述代码运行时间较长。

我们使用测试集中的所有博客进行测试。我们使用`extract_posts.py` MapReduce任务抽取博客内容，但是记得把抽取结果保存到另外一个文件夹中。

```
python extract_posts.py ~/Data/blogs_test --output-dir=/home/bob/Data/
blogposts_testing -no-output
```

回到笔记本文件，获取到输出的所有测试文档的路径。

```
testing_folder = os.path.join(os.path.expanduser("~"), "Data",
"blogposts_testing")
testing_filenames = []
for filename in os.listdir(testing_folder):
    testing_filenames.append(os.path.join(testing_folder, filename))
```

对于上面输出的每一篇测试文档，我们抽取博主性别和博客内容，然后调用预测函数进行分类。因为文档数量很多，我们不想占用太多内存，所以使用生成器来处理。生成器会给出实际性别和预测结果。

```
def nb_predict_many(model, input_filename):
    with open(input_filename) as inf:
        # remove leading and trailing whitespace
        for line in inf:
            tokens = line.split()
            actual_gender = eval(tokens[0])
            blog_post = eval(" ".join(tokens[1:]))
            yield actual_gender, nb_predict(model, blog_post)
```

然后我们记录测试集中所有数据的预测结果和实际性别。预测结果不是男性就是女性。为了

使用scikit-learn的f1_socre函数，我们需要把预测结果转换为0或1，如果结果为男性，我们记录0，反之为1。我们使用布尔测试，判断性别是否为女性，然后使用NumPy把布尔值转换为整型（int）。

```
y_true = []
y_pred = []
for actual_gender, predicted_gender in nb_predict_many(model, testing_
filenames[0]):
    y_true.append(actual_gender == "female")
    y_pred.append(predicted_gender == "female")
y_true = np.array(y_true, dtype='int')
y_pred = np.array(y_pred, dtype='int')
```

现在，我们使用scikit-learn中的F1值评估分类结果。

```
from sklearn.metrics import f1_score
print("f1={:.4f}".format(f1_score(y_true, y_pred, pos_label=None)))
```

结果为0.78，还算可以。我们使用更多的数据也许能改善分类效果。但是，我们需要使用更强大的云平台才能处理更多数据。

5. 在亚马逊EMR云平台上训练模型

我们接下来使用亚马逊的EMR（Elastic Map Reduce）云平台完成博客抽取、解析和模型创建任务。

首先，需要在亚马逊的云存储（storage cloud）创建存储段（bucket）。具体做法是用浏览器打开亚马逊S3控制界面http : //console.aws.amazon.com/s3，点击Create Bucket。记住存储段的名字，稍后会用到。

右键点击新建的存储段，选择Properties。然后，修改权限，将其设置为对所有人开放。一般来说，这样做不安全，学完这一章后，请及时修改权限。

左键点击存储段，打开它，点击Create Folder，把新文件夹命名为blogs_train。我们将把训练集数据上传到该文件夹，以便在云端处理。

我们在本地计算机上使用亚马逊的AWS CLI命令行工具操作亚马逊云平台。

使用以下命令安装该工具。

```
sudo pip2 install awscli
```

按照网址中给出的说明，为CLI工具设置安全证书：http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-set-up.html。

接下来需要上传数据到新建的存储段。首先，我们创建训练集，它包括所有除去文件名以6或7打头的文件。复制文件有很多种更为优雅的方式，但是考虑到平台兼容性，就不做推荐。相

反，我们使用最牢靠的做法，复制所有的原始文件到训练集文件夹，把文件名以6、7打头的从训练集中删除。

```
cp -R ~/Data/blogs ~/Data/blogs_train_large
rm ~/Data/blogs_train_large/6*
rm ~/Data/blogs_train_large/7*
```

接着，上传数据到亚马逊S3存储段。注意上传数据需要时间较长（好几百兆）。网速慢的读者，找个网快的地方上传比较好。

```
 aws s3 cp  ~/Data/blogs_train_large/ s3://ch12/blogs_train_large
--recursive --exclude "*" --include "*.xml"
```

接下来使用mrjob连接亚马逊EMR——它帮助我们处理所有任务；只需要给出安全证书即可。按照下面链接给出的说明，为mrjob设置安全证书：https://pythonhosted.org/mrjob/guides/emr-quickstart.html。

设置完成后，修改mrjob的运行方式，让它在亚马逊EMR上运行，改动之处很少。用`-r`开关，告诉mrjob使用`emr`。然后把输入、输出目录改为我们在S3上创建好的存储段目录。虽然使用亚马逊云平台，但是运行时间仍很长。

```
python extract_posts.py -r emr s3://ch12gender/blogs_train_large/
--output-dir=s3://ch12/blogposts_train/ --no-output
python nb_train.py -r emr s3://ch12/blogposts_train/ --output-dir=s3://
ch12/model/ --o-output
```

> 这次使用当然也是要付费的，但只有几美元[1]而已。如果你要持续运行或做其他跟大数据相关的任务，还请留意下费用问题。有一次我运行大量任务，花了20美元。我们这里任务量要小一些，估计花不到4美元。你可以查看下账户余额，设置收费提醒：https://console.aws.amazon.com/billing/home。

没必要创建`blogposts_train`和存储模型文件的文件夹——EMR会自动创建。如果这两个文件夹存在，程序反而会报错。如果你再次运行代码的话，记得把这两个文件夹改成别的名字，但是要记得同时修改两条命令中的文件名（第一条命令中的输出目录是第二条命令的输入目录）。

> 如果等待时间过长，你有些厌烦，第一项任务运行一段时间后就可以停止。我建议至少运行15分钟后再停止，可能的话，最好1小时以上。你不能中断第二项任务，中断后结果好不到哪去。第二项任务的运行时间大约是第一项任务的两到三倍。

① 澳元和美元货币符号都为$。作者表示写这本书时，两者汇率差别不大，但目前拉大了，实际费用用美元来计算比较合适。——译者注

回到S3控制界面，从存储段中下载输出的模型文件，将其保存到本地，我们就可以在笔记本文件中使用新模型。重新输入以下代码——与之前的不同之处用粗体表示，其实只更新了模型文件路径。

```
aws_model_filename = os.path.join(os.path.expanduser("~"), "models",
"aws_model")
aws_model = load_model(aws_model_filename)
y_true = []
y_pred = []
for actual_gender, predicted_gender in nb_predict_many(aws_model,
testing_filenames[0]):
    y_true.append(actual_gender == "female")
    y_pred.append(predicted_gender == "female")
y_true = np.array(y_true, dtype='int')
y_pred = np.array(y_pred, dtype='int')
print("f1={:.4f}".format(f1_score(y_true, y_pred, pos_label=None)))
```

使用更多数据后，我们发现F1值为0.81，结果较之前有所改善。

> 实验顺利完成后，如果不想继续支付费用，记得从亚马逊S3删除存储段——存储也是要钱的。

12.5　小结

本章研究的是如何运行大数据处理任务。其实，从各方面标准来看，我们的数据集还是有点小——只有几百兆。很多行业实际数据集都比这大多了，对其进行计算就需要更强大的计算能力。此外，我们所使用的算法可根据具体的任务做进一步优化，以提升扩展能力。

为了预测博主的性别，我们从博客文章中抽取词频特征。我们借助mrjob包，用映射和规约方法抽取博客内容和词频。抽取完成后，训练朴素贝叶斯分类器，预测博主性别。

我们可以用mrjob包先在本地测试，然后使用亚马逊的EMR云平台。你也可以使用其他云平台，甚至定制亚马逊EMR集群运行MapReduce任务，但是需要更多的操作才能跑起来。

附 录

接下来的方向

全书正文部分至此已结束，学习本书的过程中，也许你已经注意到有很多路我们没有走，遇到多种选择的情况，我们可能只讲了其中几种，还有些主题没有充分展开。在附录部分，我整理了各章进一步学习方向，便于那些意犹未尽的读者深入学习，继续提升用Python挖掘数据的能力。如果你乐于接受挑战的话，本章不容错过。

附录按照章节来组织，每章给出与数据挖掘相关的文章、书籍或其他资源，同时还就该章挖掘任务提出新的挑战，有时只让读者做些小改进，但也给出了几个工作量比较大的挑战——对于难度较大的任务，介绍内容也会更多一些。

第 1 章——开始数据挖掘之旅

Scikit-learn 教程

http://scikit-learn.org/stable/tutorial/index.html

scikit-learn文档提供了一系列数据挖掘教程。教程涵盖范围广，从适合新手把玩的数据集一直介绍到最近研究所用的技术。

想从头到尾学一遍这个教程，不肯下功夫是不行的——这份教程很全面——也值得在上面下功夫。

扩展 IPython Notebook

http://ipython.org/ipython-doc/1/interactive/public_server.html

IPython Notebook功能强大，扩展方式很多，比如可以在别的计算机上创建服务器，运行笔记本文件，在你的主计算机上访问即可。如果你的主计算机性能一般，比如配置还算凑合的笔记本电脑，而其他可以供你支配的计算机性能很好，使用IPython Notebook在性能不错的计算机运行服务器是个好主意。此外，你还可以创建节点，采用并行方式处理数据。下面网站有不少数据

集，一并分享给你。

http://archive.ics.uci.edu/ml/。

网上有很多由学术、商业和政府机构提供的数据集。UCI机器学习数据库里有一系列标注好的数据集，用它们来测试算法很不错。

尝试下看看OneR算法在不同数据集上的表现。

第2章——用 `scikit-learn` 估计器分类

k近邻算法的扩展

https://github.com/jnothman/scikit-learn/tree/pr2532

k-近邻算法的简单实现速度很慢——它查看任意两个数据点是否足够近。当然还有更好的方法，`scikit-learn`实现了几种.。例如，我们可以用kd-tree提升算法速度。

另外一种提升搜索速度的方法叫作*局部敏感散列*（Locality-Sensitive Hashing，LSH）。这是对`scikit-learn`提出的一项改进，写作本书时，还没有加到`scikit`库中。上述链接为`scikit-learn`库的一个开发分支，你可以在数据集上测试LSH。具体方法还请参考该分支的相关文档。

复制代码仓库，按照下述网址给出的方法，安装最新版本：http://scikit-learn.org/stable/install.html。

记住要使用这个Git仓库的代码，而不要使用官方的代码。我建议你使用`virtualenv`或虚拟机安装，不要直接将其装到你的计算机上。这里有一份很棒的`virtualenv`教程：http://docs.python-guide.org/en/latest/dev/virtualenvs/。

更多复杂的流水线

http://scikit-learn.org/stable/modules/pipeline.html#featureunion-composite-feature-spaces

书中所用到的流水线都是线性的——上一步的输出作为下一步的输入。

流水线也遵循转换器和估计器的接口——流水线可以嵌套，方便构造更复杂的模型，再结合使用上述链接提到的FeatureUnion[①]方法，流水线将变得更加强大。

① FeatureUnion：将多个转换器对象结合在一起，组成一个新的转换器，汇总每个转换器的输出结果。——译者注

这样一次可以抽取多种类型的特征，组成新的数据集。更多细节及示例请见：http://scikit-learn.org/stable/auto_examples/feature_stacker.html。

比较分类器

`scikit-learn`提供了很多分类器。在选择分类器时，我们需要考虑一系列因素。你可以通过比较它们的F1值确定哪种效果更好。通过计算各F1值的标准差，就能确定几个分类器的分类效果之间的差异是否显著。

另外需要注意的是，几个分类器应该使用相同的训练集和测试集——也就是说，第一个分类器的测试数据，应该跟其他所有分类器的测试数据相同。使用相同的随机状态可以保证这一点——对于重现实验结果很重要。

第 3 章——用决策树预测获胜球队

pandas 的更多内容

http://pandas.pydata.org/pandas-docs/stable/tutorials.html

pandas库很周到——你加载数据时所能用到的各种功能，它很可能都已实现了。更多内容请见上面的链接。

Chris Moffitt 写了篇很不错的博文，介绍常见的Excel数据处理任务，如何用pandas完成：http://pbpython.com/excel-pandas-comp.html。

你还可以用pandas处理大数据集；可以看下StackOverflow用户Jeff对以下问题的解答（写作本书时[①]是最佳答案），了解处理过程：http://stackoverflow.com/questions/14262433/large-data-work-flows-using-pandas。

Brian Connelly的pandas教程也很棒：

http://bconnelly.net/2013/10/summarizing-data-in-python-with-pandas/。

更多复杂特征

http://www.basketball-reference.com/teams/ORL/2014_roster_status.html

① 翻译本书时依旧是最佳答案。——译者注

每个赛季不同的比赛，上场队员也会有所不同。本来志在必得的比赛，很可能因为核心球员受伤而打得很艰难。你可以从篮球参考网站找到球队的队员名单。例如，上述链接指向的就是2013—2014赛季奥兰多魔术队（Orlando Magic）球员名单——所有NBA球队的类似数据都能从该网站找到。

编写代码，整合队员异动信息，以此作为新特征，可以有效提升模型表现，但是工作量不小！

第 4 章——用亲和性分析方法推荐电影

新数据集

http://www2.informatik.uni-freiburg.de/~cziegler/BX/

很多用于特定领域推荐任务的数据集，都值得探索。例如，Book-Crossing数据集包含来自27.8万用户的100多万条图书打分信息。有些评价信息很明确（用户给出分数），而有些则很模糊。模糊评价的权重不应该跟明确评价的一样高。

音乐网站www.last.fm公开了可用于音乐推荐的数据集：http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/。

这里有一个用于笑话推荐的数据集！请见http://eigentaste.berkeley.edu/dataset/。

Eclat 算法

http://www.borgelt.net/eclat.html

本章实现的Apriori算法是最常用的关联规则挖掘算法，但不是最好的。比较而言，Eclat算法更年轻也更强大，实现起来也比较容易。

第 5 章——用转换器抽取特征

增加噪音

本章讲到了如何通过*删除*噪音来提升性能；然而，有些数据集需要*增加*噪音。原因很简单——防止分类器过拟合训练数据，从而生成适用性更强的规则（但过多的噪音会导致模型过于宽泛，效果反而不好）。尝试实现往数据集中添加噪音的转换器。在UCI ML的数据集上进行测试，看看能否提升在测试集上的表现。

Vowpal Wabbit

http://hunch.net/~vw/

Vowpal Wabbit项目用于快速从文本中抽取特征。它针对Python进行了封装，可以直接在Python代码中调用。在大数据集上试试它的效果，比如第12章使用的博客语料。

第 6 章——使用朴素贝叶斯进行社会媒体挖掘[1]

垃圾信息监测

http://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter

我们用本章所学到的内容，就可以创建用来监测社会媒体广播是否为垃圾消息的应用。你可以尝试创建由垃圾广播/正常广播组成的数据集，实现文本挖掘算法，评估结果。

垃圾监测算法需要考虑假阳性和假阴性的情况。很多人宁愿多浏览几条垃圾信息，也不愿因为垃圾过滤器过于强大而错过合法的信息。为了满足人们这一需求，你可以把F1值作为评估标准，用网格搜索寻找合适的参数值，具体方法请见上面链接。

自然语言处理和词性标注

http://www.nltk.org/book/ch05.html

本章我们所用到的技术比起其他领域使用的复杂语言模型算是轻量级。例如，别的模型可能使用词性标注来消除同形异义词的歧义，提升准确性。NLTK配套的书[2]有一章专门讲这个问题，详细内容请见上面的链接。全书也很值得一读。

第 7 章——用图挖掘找到感兴趣的人

更复杂的算法

https://www.cs.cornell.edu/home/kleinber/link-pred.pdf

① 社会媒体挖掘相关技术可参考人民邮电出版社出版的《社会媒体挖掘》一书。——编者注

② 中文版书名为《Python自然语言处理》，国内还可以找到英文版的影印版。据NLTK官网介绍2016年上半年将出版第2版。——译者注

关于如何预测包括社交网络在内的图结构中节点之间是否存在边，已经有大量相关研究。例如，David Liben-Nowell和Jon Kleinberg曾就这一主题发表了一篇论文，对于更复杂算法的设计很有帮助，论文请见上面的链接。

NetworkX

https://networkx.github.io/

如果你以后经常处理图和网络结构，深入研究NetworkX很有必要——可视化选项很好用，算法实现也非常不错。另外一个叫作SNAP的库也针对Python做了封装，请见http://snap.stanford.edu/snappy/index.html。

第 8 章——用神经网络破解验证码

好（坏？）验证码

http://scikit-image.org/docs/dev/auto_examples/applications/plot_geometric.html

我们所破解的验证码比起如今网站常用的要简单不少。你可以使用以下技巧来提升破解难度：

- 采用不同的转换方法。比如`scikit-image`所提供的（请见上面链接）
- 使用不同的颜色；尝试无法很好转换为灰度模式的颜色
- 在图像中添加线条或其他形状：http://scikit-image.org/docs/dev/api/skimage.draw.html

深度网络

使用上述技巧产生的验证码会愚弄我们的破解工具，我们只好改进它。尝试使用第11章的深度网络。

网络越大，需要更多数据，因此你可能需要生成几千张验证码图像以达到好的效果。数据集的生成适合用并行方式来处理——因为它包含大量可以被分别处理的小任务。

增强学习

http://pybrain.org/docs/tutorial/reinforcement-learning.html

数据挖掘领域的下一个方向——增强学习势头日猛，虽然它的出现已经有一段时间了！

PyBrain提供了几种增强学习算法，值得在刚创建的验证码数据集上一试（其他数据集也行！）。

第 9 章——作者归属问题

增加数据量

本章预测发件人的应用只使用了一部分邮件，数据集中还有很多邮件可以用。增加发件人的数量可能会导致正确率下降，但是使用相似的方法也有可能提升算法的表现。N元语法中的N以及参数到底取什么值，支持向量机能达到最佳效果，使用网格搜索就能解决这两个问题。我们争取在发件人数量很多的情况下，也能取得好的效果。

博客语料

第12章博客语料给出了博主编号（每个编号代表一个博主），把博主编号看成是类别，就能将其用作本章实验语料。此外，性别、年龄、行业和星座都可以作为分类任务来进行预测——解决作者归属的方法擅长处理这些分类问题吗？

局部 N 元语法

https://github.com/robertlayton/authorship_tutorials/blob/master/LNGTutorial.ipynb

另外一种分类器使用局部N元语法，从每个作者的作品而不是全部作者的作品中选择最佳特征。我写了一篇用局部N元语法解决作者归属问题的教程，请见上面链接。

第 10 章——新闻语料分类

算法评价

聚类算法效果评价是个大难题——一方面，我们也能多少知道好的分簇结果应该是什么样子；另一方面，如果我们确实知道，那就应该标注一部分数据，使用有监督的分类器对数据进行分类！该话题相关文章很多。下面这份文档介绍了聚类算法评价的难点所在：

http://www.cs.kent.edu/~jin/DM08/ClusterValidation.pdf

这里有份关于该主题的更全面（虽然有点过时）的一份文档：http://web.itu.edu.tr/sgunduz/courses/verimaden/paper/validity_survey.pdf。

`scikit-learn`实现了几种评价方法，对它们的综述请见http://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation。

有了这些方法，你就能评估使用哪些参数值可以取得更好的聚类效果。网格搜索能帮助我们找到参数的最佳取值——就像在分类中用到的那样！

近期趋势分析

你可以运行我们本章编写的代码几个月。为期间发现的每个簇添加几个标签，便于我们了解哪些主题一直很活跃，从而了解近几个月全世界新闻焦点。

我们可以使用调整互信息等评价标准，比较聚类结果，请见上面评价方法综述链接。观察下一个月、两个月、半年和一年之后聚类的变化结果。

实时聚类

k-means算法可以随着时间的推进，不断训练和更新，而不是在给定时间内做离散分析。你可以通过几种方法跟踪聚类变化——例如，跟踪每个簇中频率最高的几个单词或质心点每天移动的距离。别忘了网站API使用上限——你可能只需要每隔几个小时检查下有无新数据，就能保证结果是实时的。

第 11 章——用深度学习方法为图像中的物体进行分类

Keras 和 Pylearn2

如果你想进一步了解如何用Python做深度学习，Keras和Pylearn2这两个库有必要了解一下。它们都是以`Theano`为基础，但各有各的使用方法和特点。

Keras请见https://github.com/fchollet/keras/。

Pylearn2请见http://deeplearning.net/software/pylearn2/。

写作本书时，这两个库还不成熟，比较起来Pylearn2更稳定。它们都能很好完成你交给它们的任务，在做其他深度学习项目时，可以尝试用下。

另外一个很火的库叫Torch，但是写作本书时，还没有Python接口可用（请见http://torch.ch/）。

Mahotas

Mahotas图像处理包提供更好和更复杂的图像处理方法，使用它们能提升正确率，虽然计算量可能大幅上升。然而，很多图像处理任务适合并行处理，所以计算量加大也不再是问题。图像分类相关研究资料很多，下面这两份文档给出不少资料，可以先从这里找起：

http://luispedro.org/software/mahotas/.

http://ijarcce.com/upload/january/22-A%20Survey%20on%20Image%20Classification.pdf

其他图像数据集有：

http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

一些学术机构和行业组织也开放了很多图像数据集。上述网站介绍了很多数据集及使用它们的最好算法。要实现这样的一些好算法需要大量代码，但是回报远大于付出。

第 12 章——大数据处理

Hadoop 课程

雅虎和谷歌面向不同水平的开发者提供了一系列非常实用的Hadoop课程。虽然侧重点不在Python，但是了解Hadoop概念后，再把它们应用到Pydoop或其他类似的库都能取得很好的效果。

雅虎的课程：https://developer.yahoo.com/hadoop/tutorial/。

谷歌的课程：https://cloud.google.com/hadoop/what-is-hadoop。

Pydoop

Pydoop是一个用来运行Hadoop任务的Python库——教程请见http://crs4.github.io/pydoop/tutorial/index.html。

Pydoop可以跟HDFS配合使用，mrjob有类似功能，但在运行某些任务时，Pydoop能给你更多的控制权。

推荐引擎

创建大型的推荐引擎是测试你大数据技能的好方法。Mark Litwintschik在他的博文中介绍了如何使用大数据技术Apache Spark创建推荐引擎，详见http://tech.marksblogg.com/recommendation-engine-spark-python.html。

更多资源

Kaggle竞赛：www.kaggle.com/

Kaggle定期举行数据挖掘竞赛，一般都会提供奖金。参加Kaggle竞赛是学习处理真实场景数据挖掘任务的好方法。他们的社区用户友善、乐于分享——通常你能见到竞赛前10名选手分享代码！

Coursera在线学习网站：

www.coursera.org

Coursera网站上有很多关于数据挖掘和数据科学的课程，大多是专注于某个方向，比如有专门讲大数据和图像处理的。吴恩达（Andrew Ng）老师的机器学习课程综合性强，涉及面比较广，这门课程很受欢迎，从它开始学起很不错：https://www.coursera.org/learn/machine-learning/。

这门课所讲的内容比本书难度要大一点，感兴趣的读者可以将其作为今后的学习方向。

神经网络方面，请留意下这门课程：https://www.coursera.org/course/neuralnets。

如果你完成了上述课程的学习，可以关注下这门关于概率图模型的课程：https://www.coursera.org/course/pgm。

在数据规模急速膨胀的大数据时代，数据挖掘这项甄别重要数据的核心技术正发挥越来越重要的作用。它将赋予你解决实际问题的“超能力”：预测体育赛事结果、精确投放广告、根据作品的风格解决作者归属问题，等等。

本书使用简单易学且拥有丰富第三方库和良好社区氛围的Python语言，由浅入深，以真实数据作为研究对象，真刀实枪地向读者介绍Python数据挖掘的实现方法。通过本书，读者将迈入数据挖掘的殿堂，透彻理解数据挖掘基础知识，掌握解决数据挖掘实际问题的最佳实践！

- 理解决策树、朴素贝叶斯、支持向量机和深度学习
- 运用常见算法为解决现实问题建立数据模型
- 利用API从Reddit等网站获取数据集
- 从数据集中找出并提取特征
- 使用数据集设计并开发数据挖掘应用
- 基于实时数据，进行大数据处理

★ 亚马逊读者评论

★“不错的数据挖掘读物。浅显易懂，是遇到类似问题时的绝佳参考书籍。”

★“本书用简单通俗的语言讲解数据挖掘，并附有大量代码示例。强烈推荐给Python的新手用户和狂热爱好者。”

[PACKT] PUBLISHING

图灵社区：iTuring.cn
热线：(010) 51095186 转 600

分类建议 计算机科学 / 大数据与数据挖掘

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-42710-6

ISBN 978-7-115-42710-6
定价：59.00元